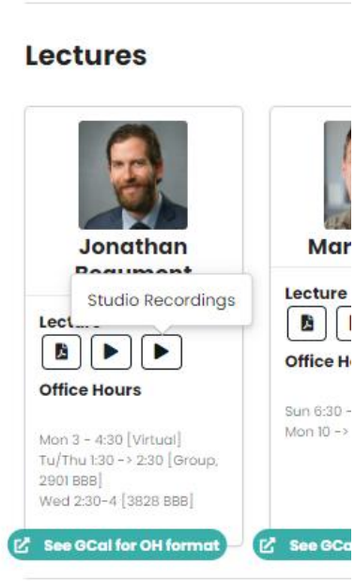
EECS 370 - Lecture 12

Multi-cycle +
Introduction to Pipelining

Reminder

- If you're watching lectures asynchronously…
- I have studio recordings
  - Much better quality than lecture recordings
  - I won't walk off screen, etc



Extra Lecture Section

- Professor Gokul has moved his lecture section to 10:30 am in EWRE
  - Feel free to attend those if that time works better for you

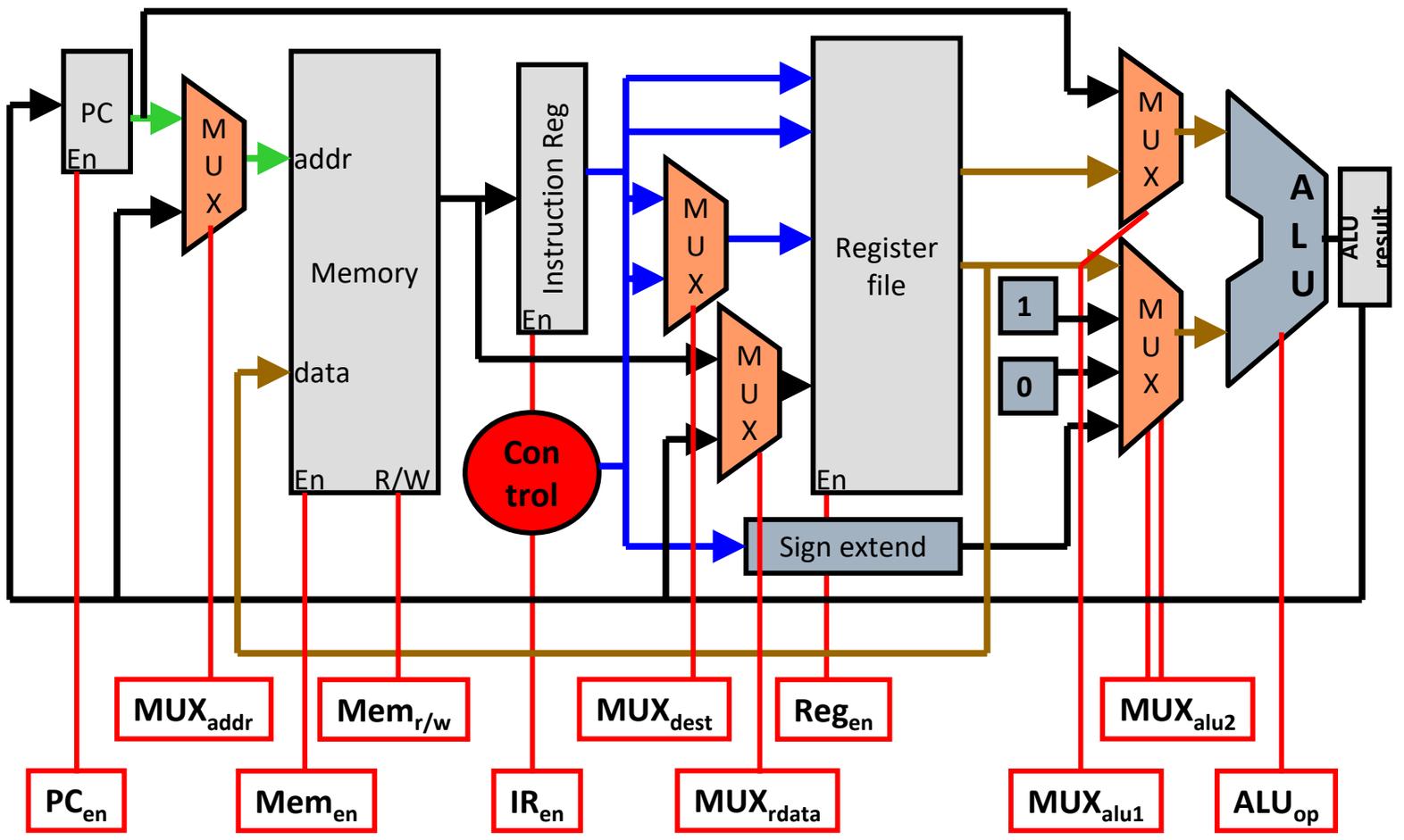
Announcements

- P2
  - P2a due Thursday
  - P2L due after break
- HW 2
  - Posted, due next week
- Lab 6 on Fri / Mon
- Midterm Exam
  - Thu Oct 12th, 6-8 pm (next week)
  - Sample exams on website
  - You can bring 1 sheet (double sided is fine) of notes
  - We will provide LC2K encodings + ARM cheat sheet

Review: What's Wrong with Single-Cycle?

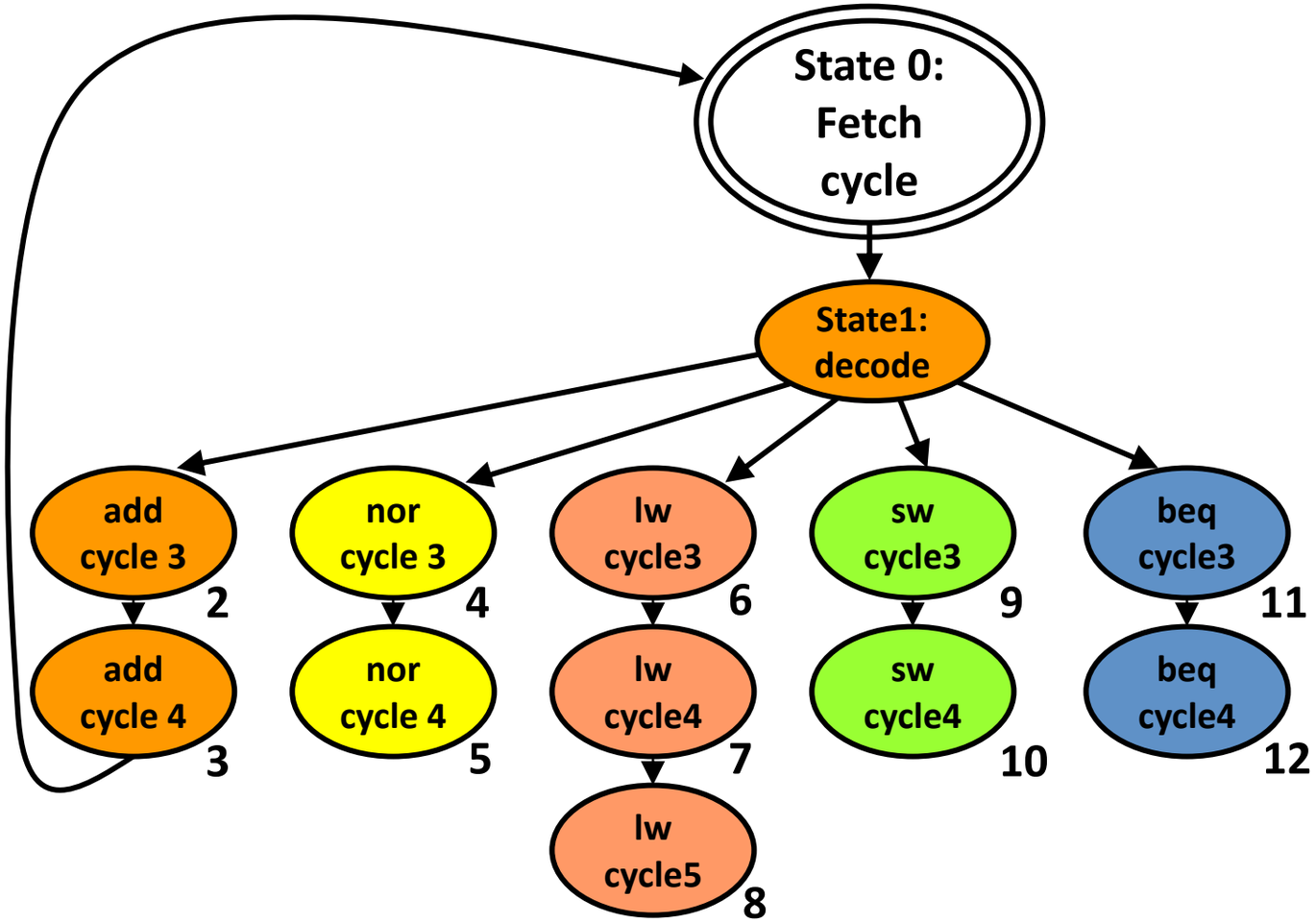
- **All instructions run at the speed of the slowest instruction.**
- Adding a long instruction can hurt performance
  - What if you wanted to include multiply?
- You cannot reuse any parts of the processor
  - We have 3 different adders to calculate PC+1, PC+1+offset and the ALU
- No benefit in making the common case fast
  - Since every instruction runs at the slowest instruction speed
    - This is particularly important for loads as we will see later

Review Multi-cycle LC2K Datapath



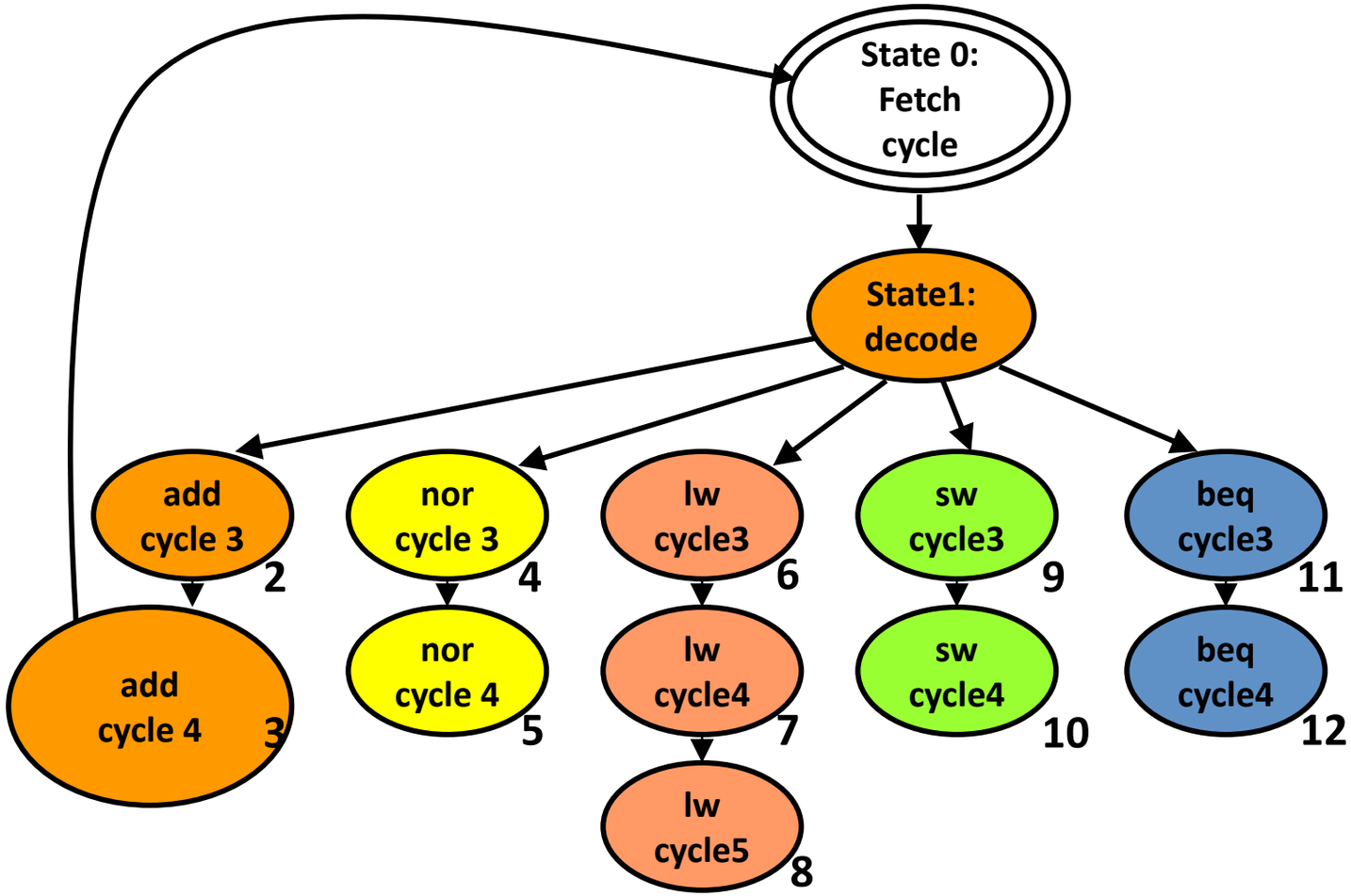
Each red signal comes from "Control" (implemented via ROM as before)

State machine for multi-cycle control signals (transition functions)

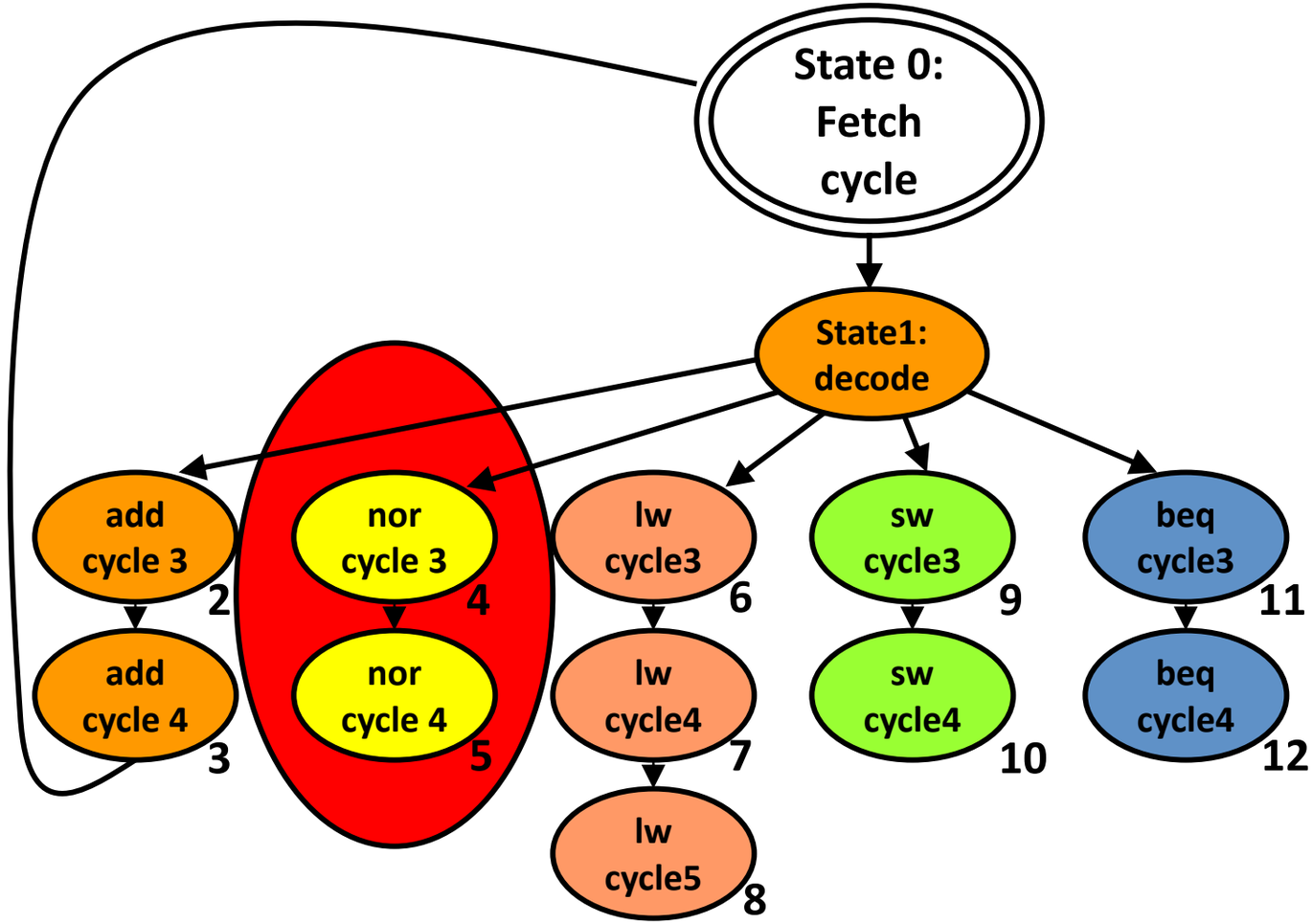


Note: we aren't worrying about JALR instruction in hardware going forward

State 3: Add cycle 4

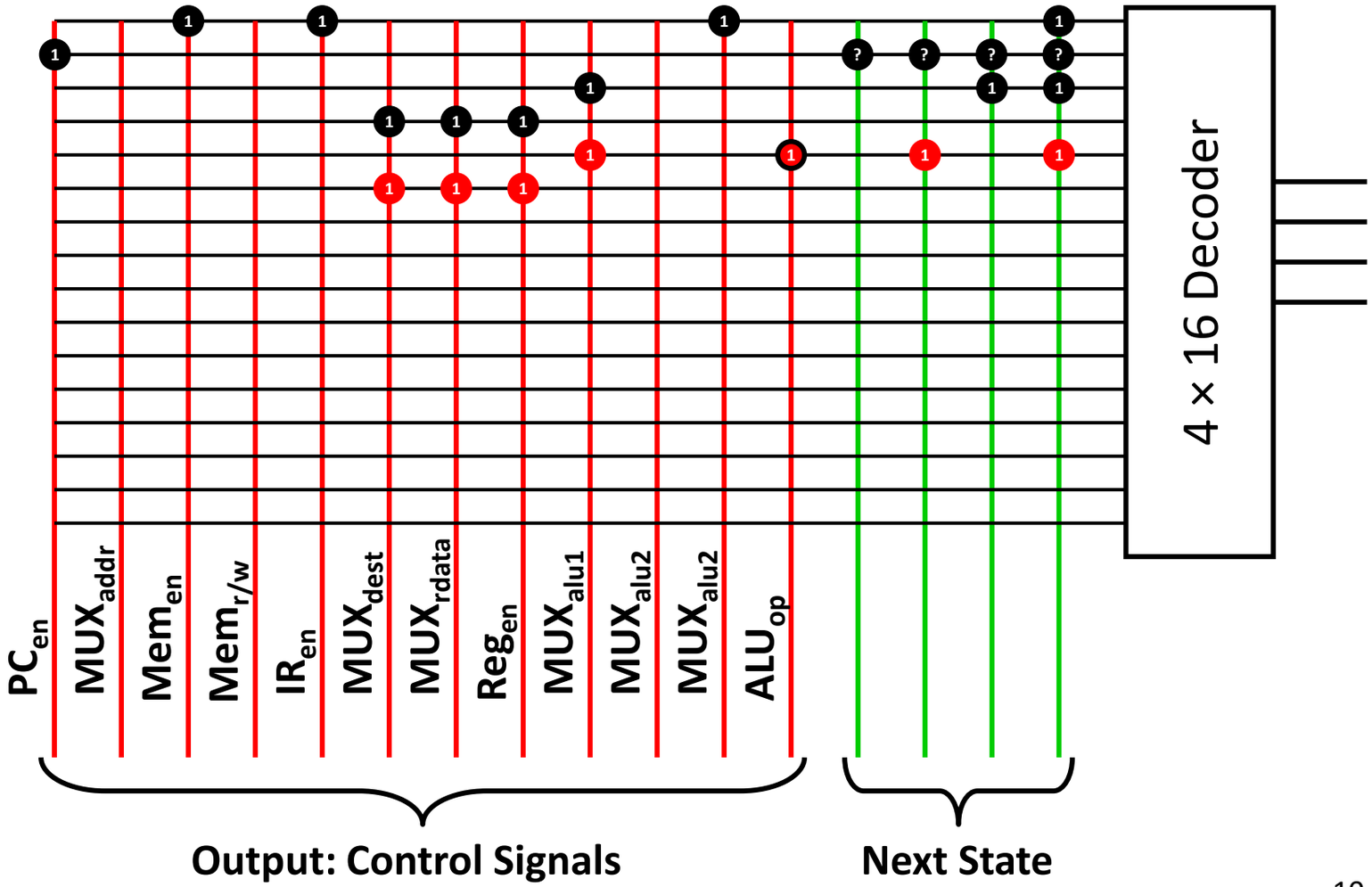


Return to State 0: Fetch cycle to execute the next instruction

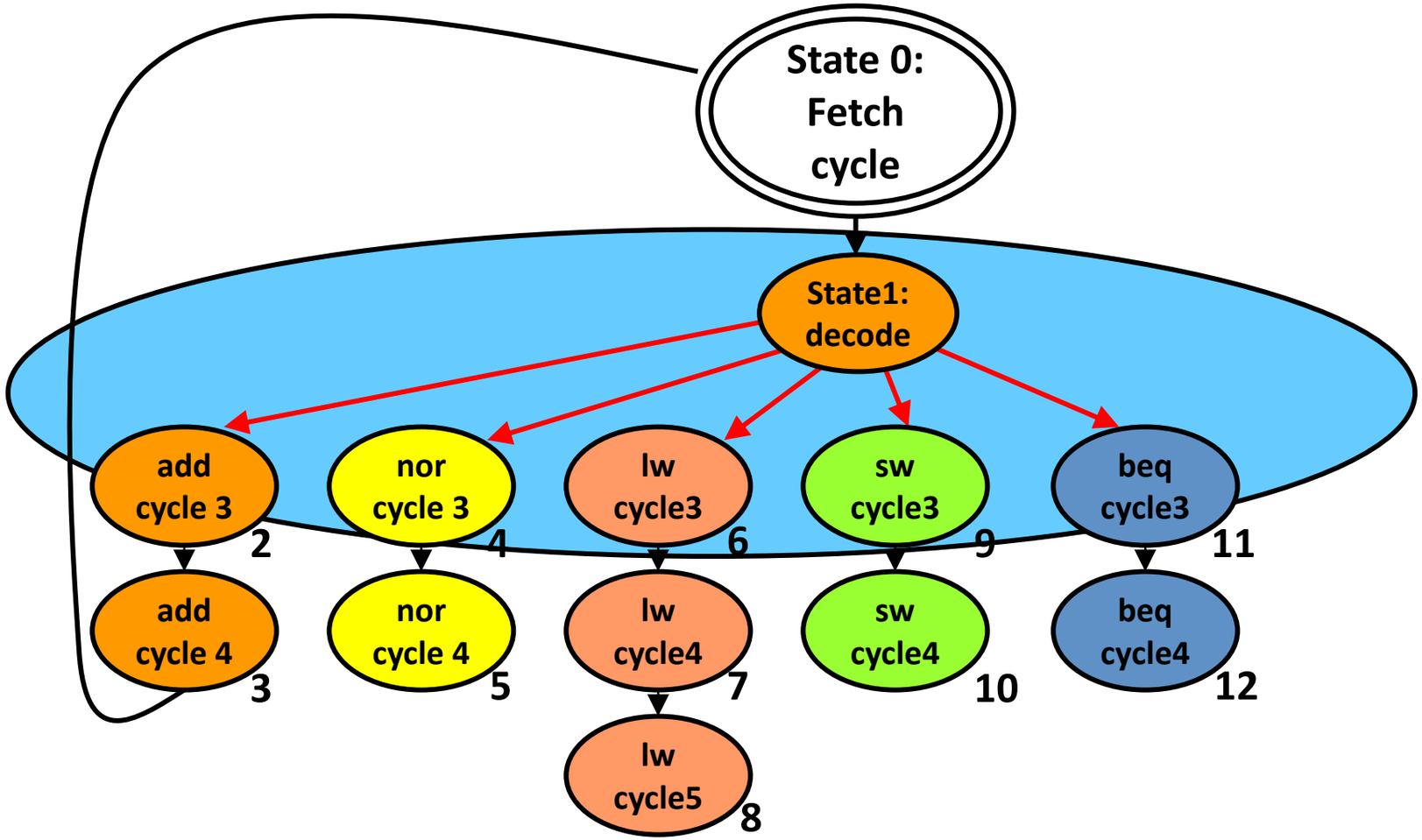


Control Rom for nor (4 and 5)

Same output as add except ALU_{op} and Next State

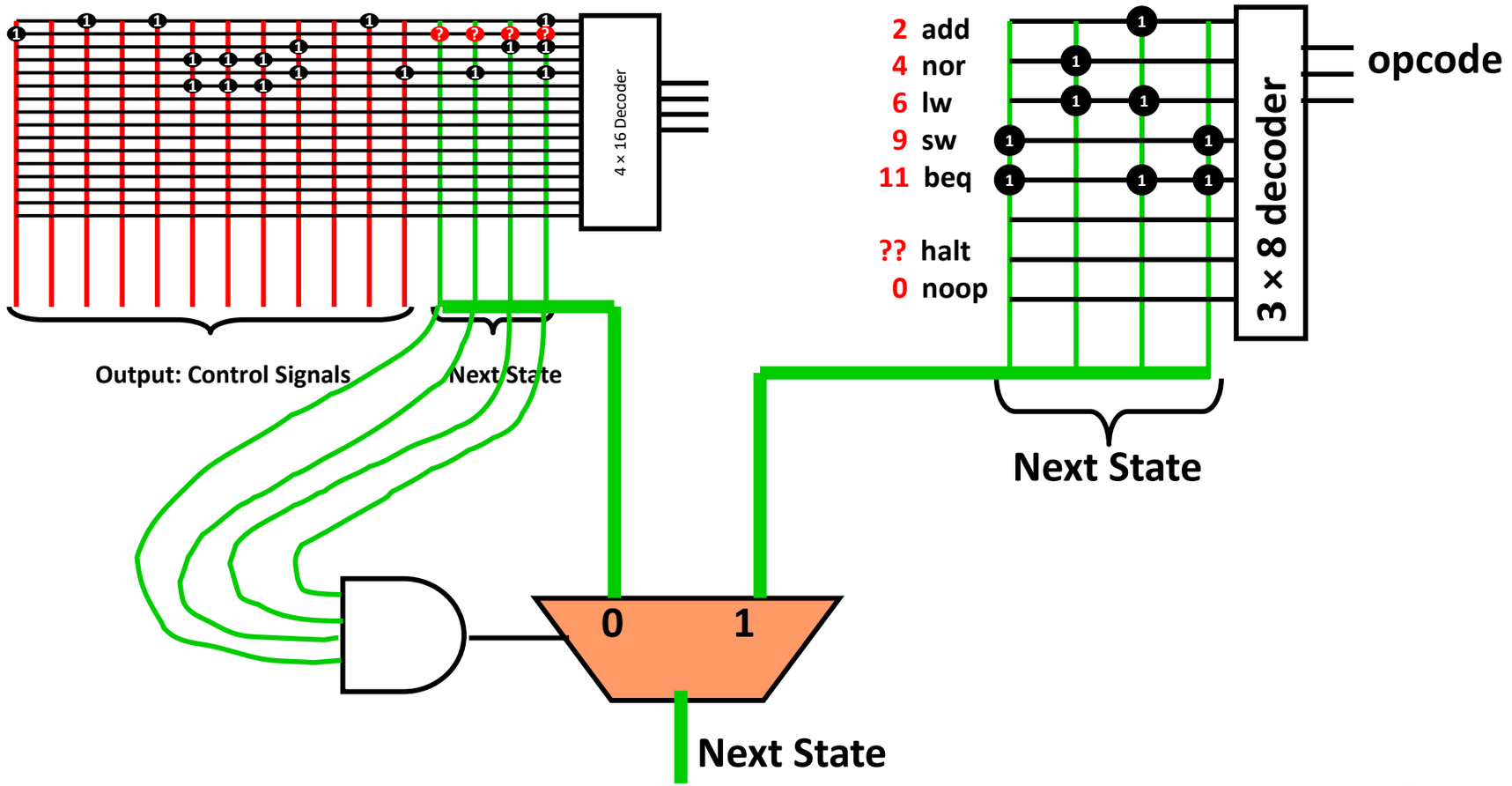


What about the transition from state 1?

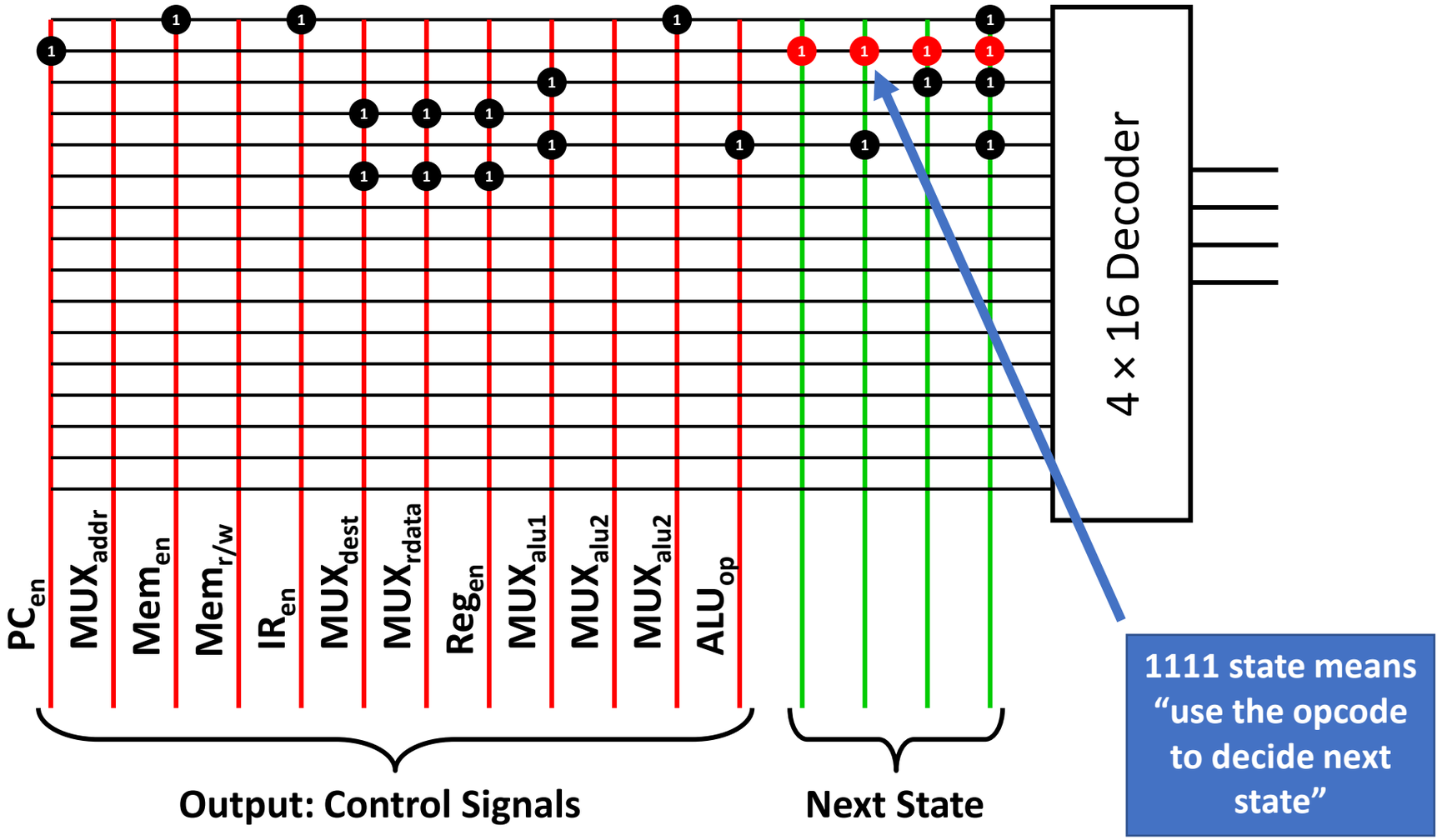


Complete transition function circuit

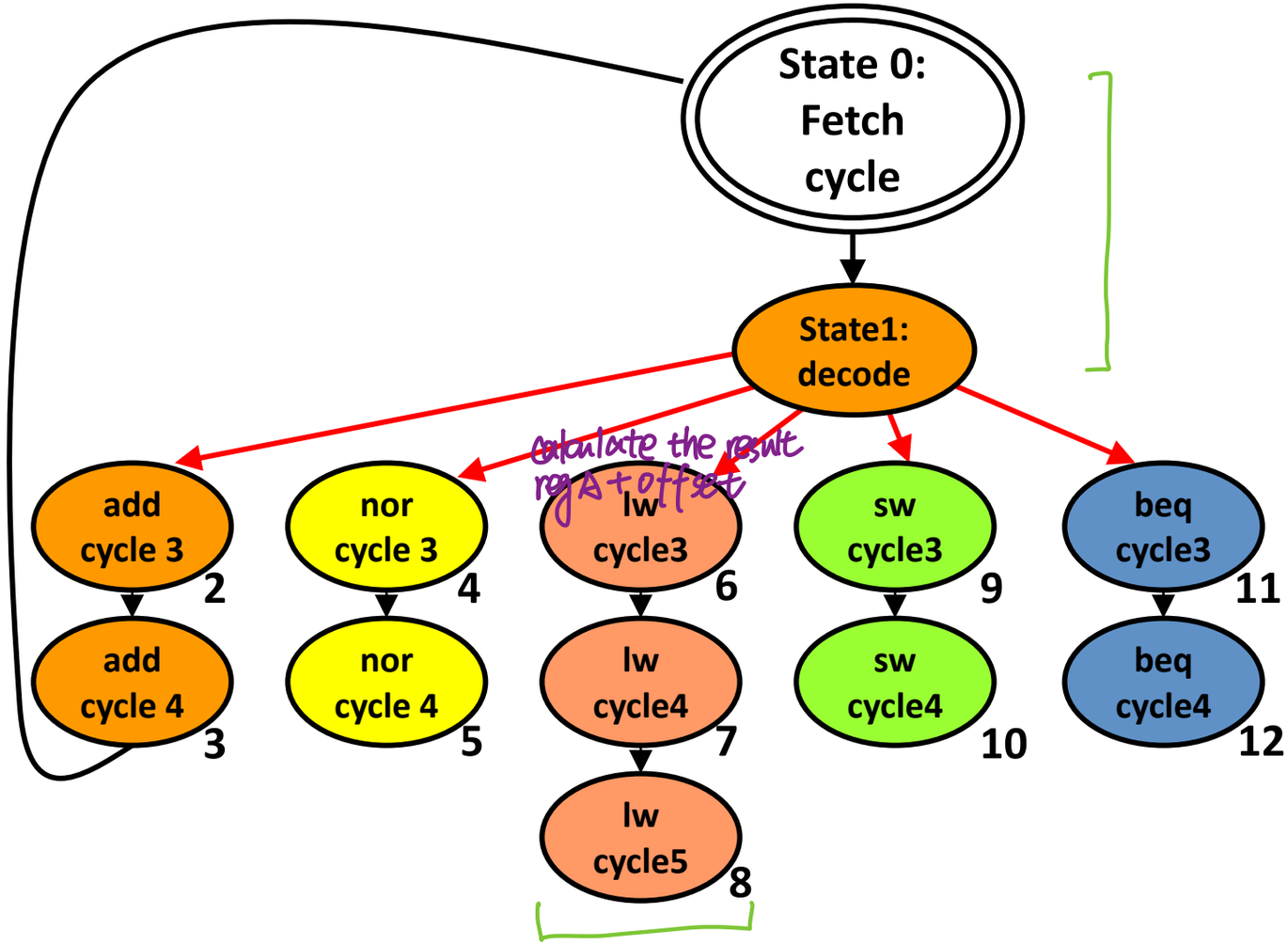
Secondary ROM stores which state we should branch to after decode for each opcode



Control Rom (use of 1111 state)

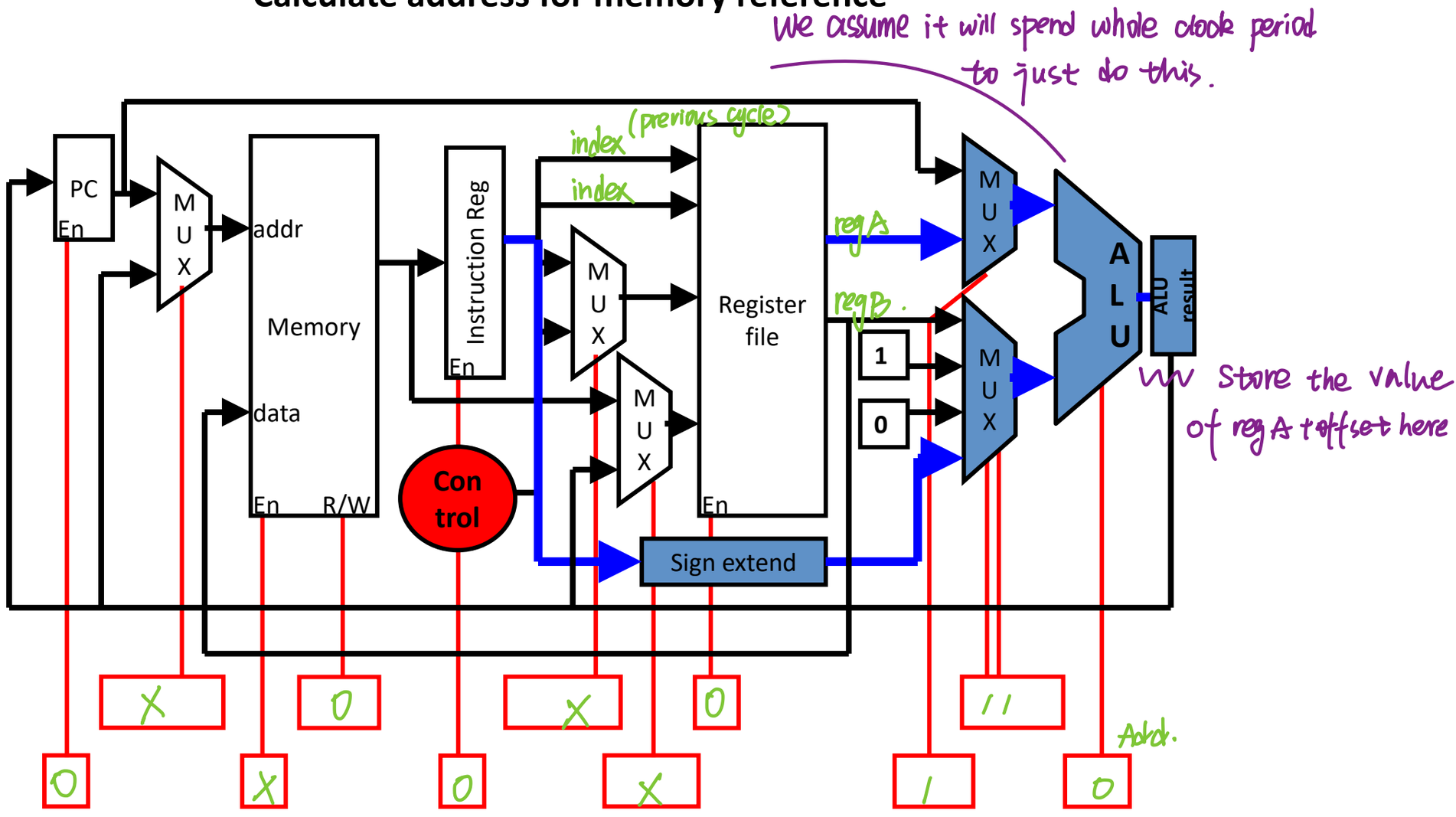


Return to State 0: Fetch cycle to execute the next instruction



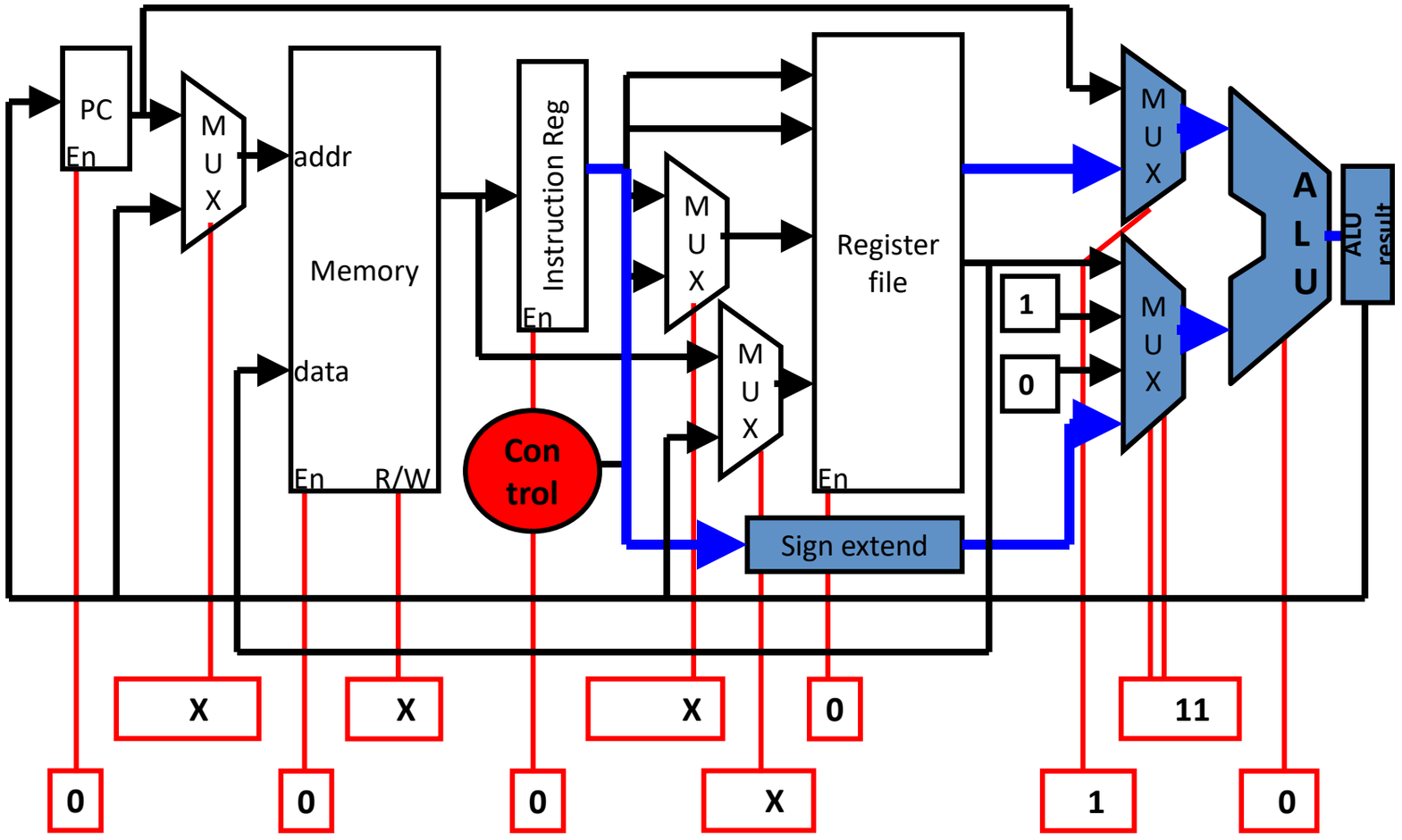
State 6: LW cycle 3

Calculate address for memory reference

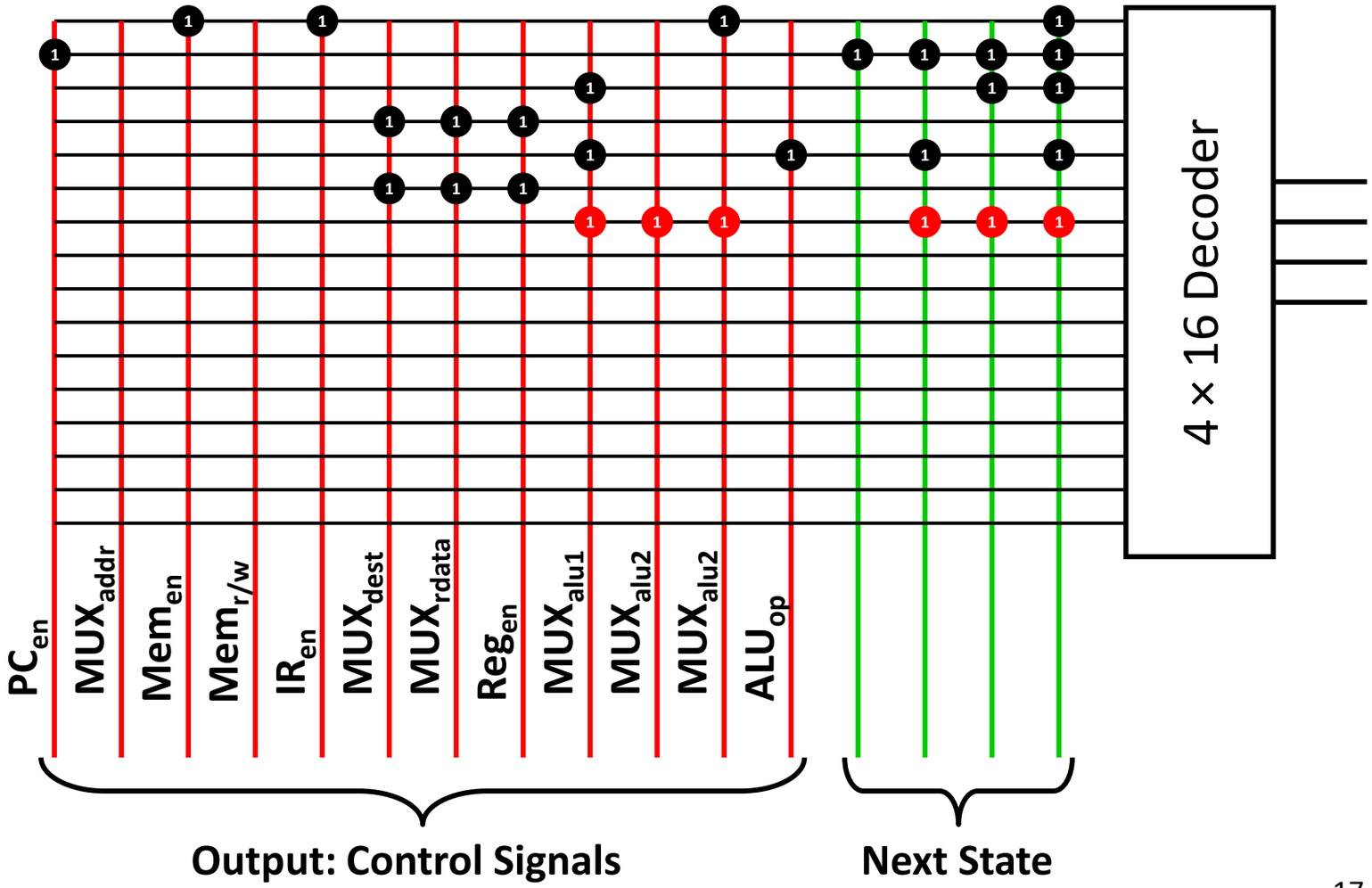


State 6: LW cycle 3

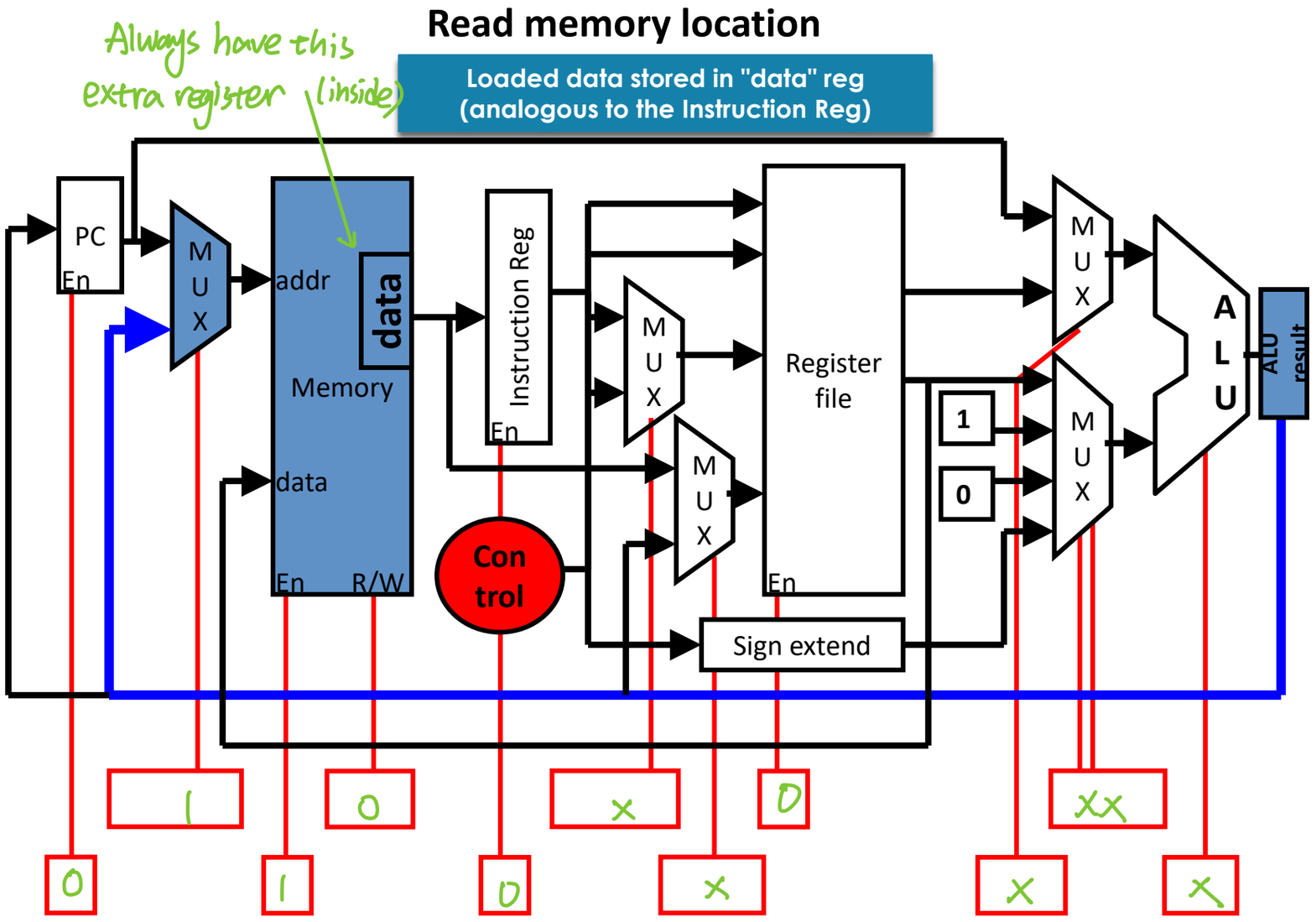
Calculate address for memory reference



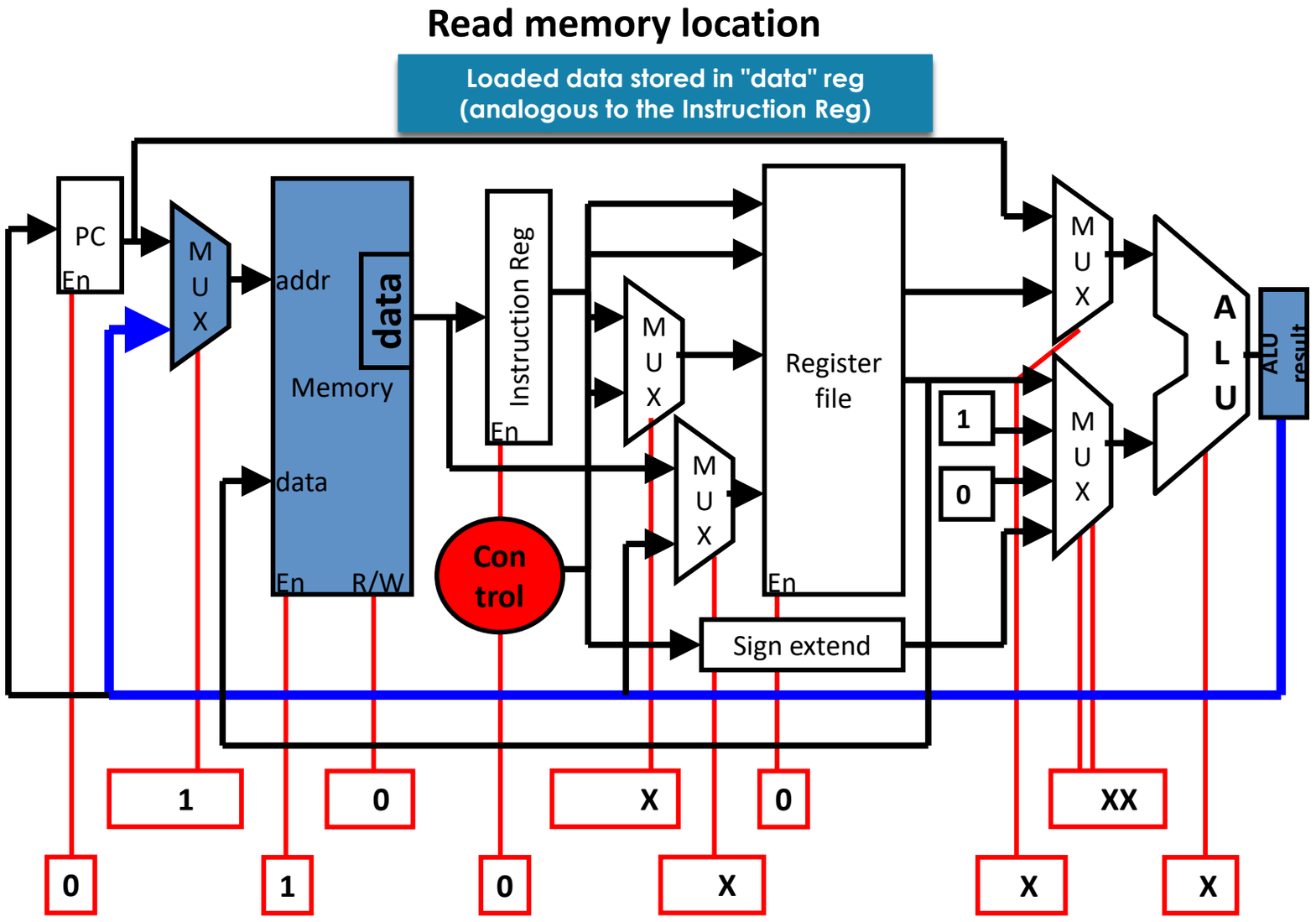
Control Rom (lw cycle 3)



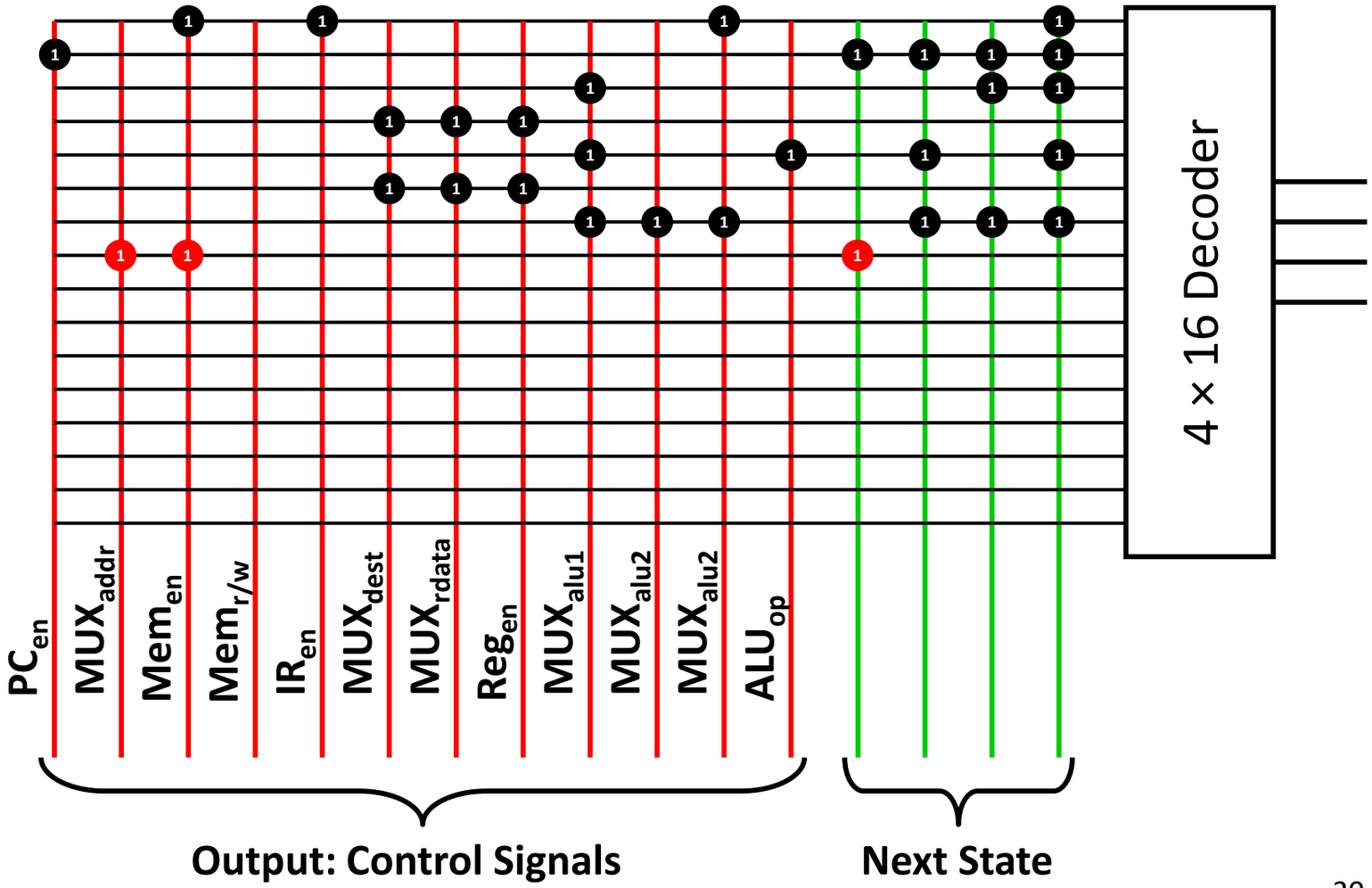
State 7: LW cycle 4



State 7: LW cycle 4

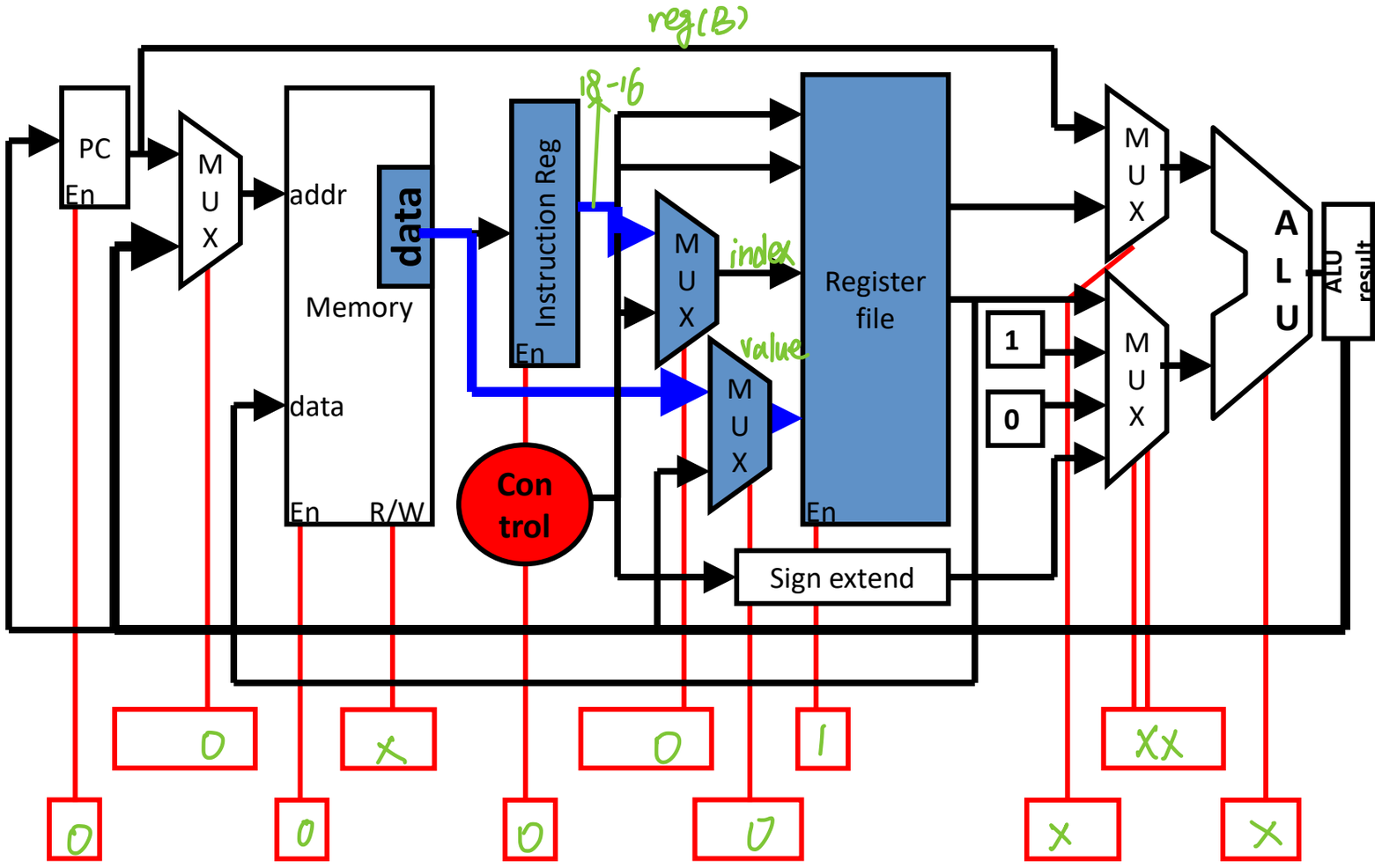


Control Rom (lw cycle 4)



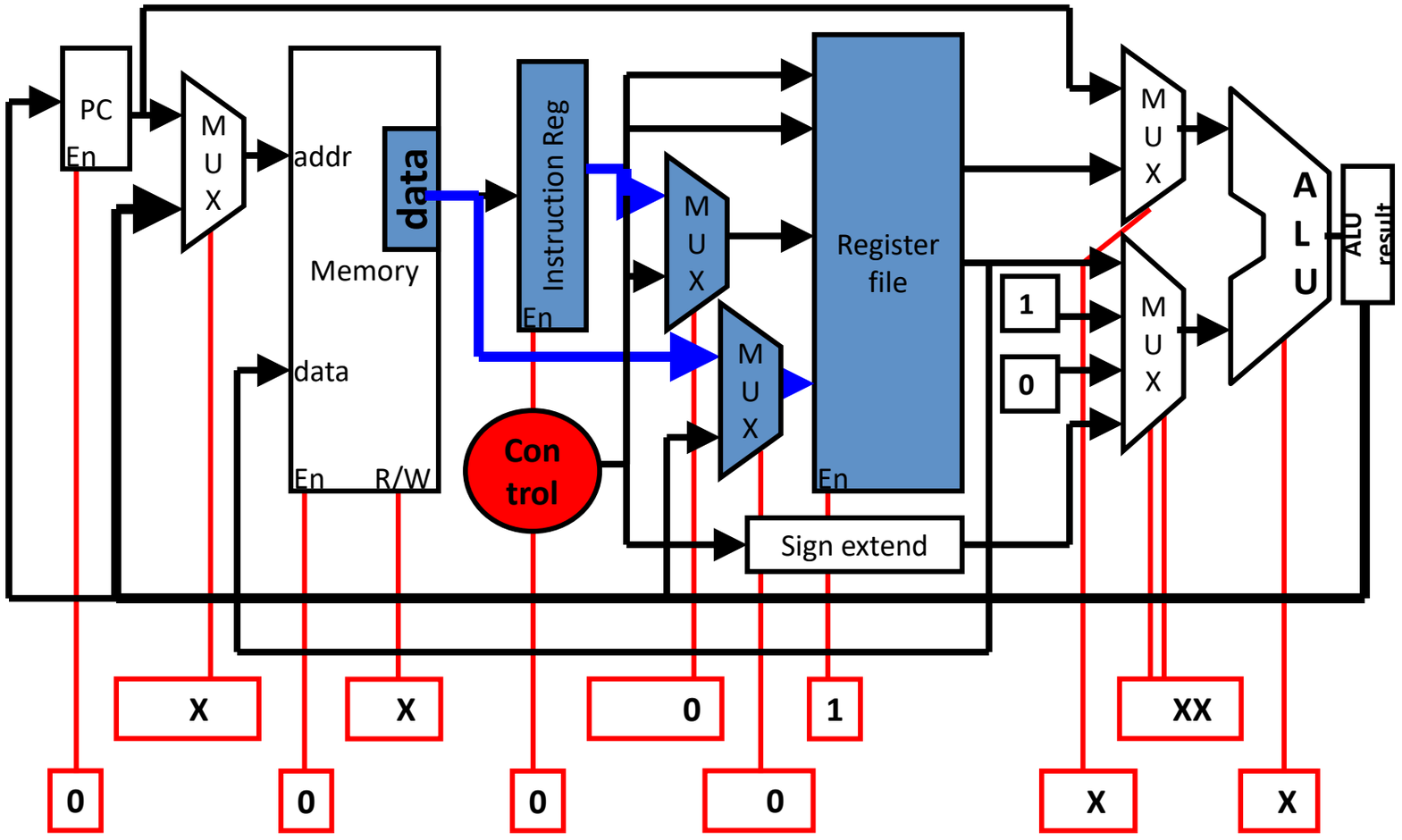
State 8: LW cycle 5

Write memory value to register file

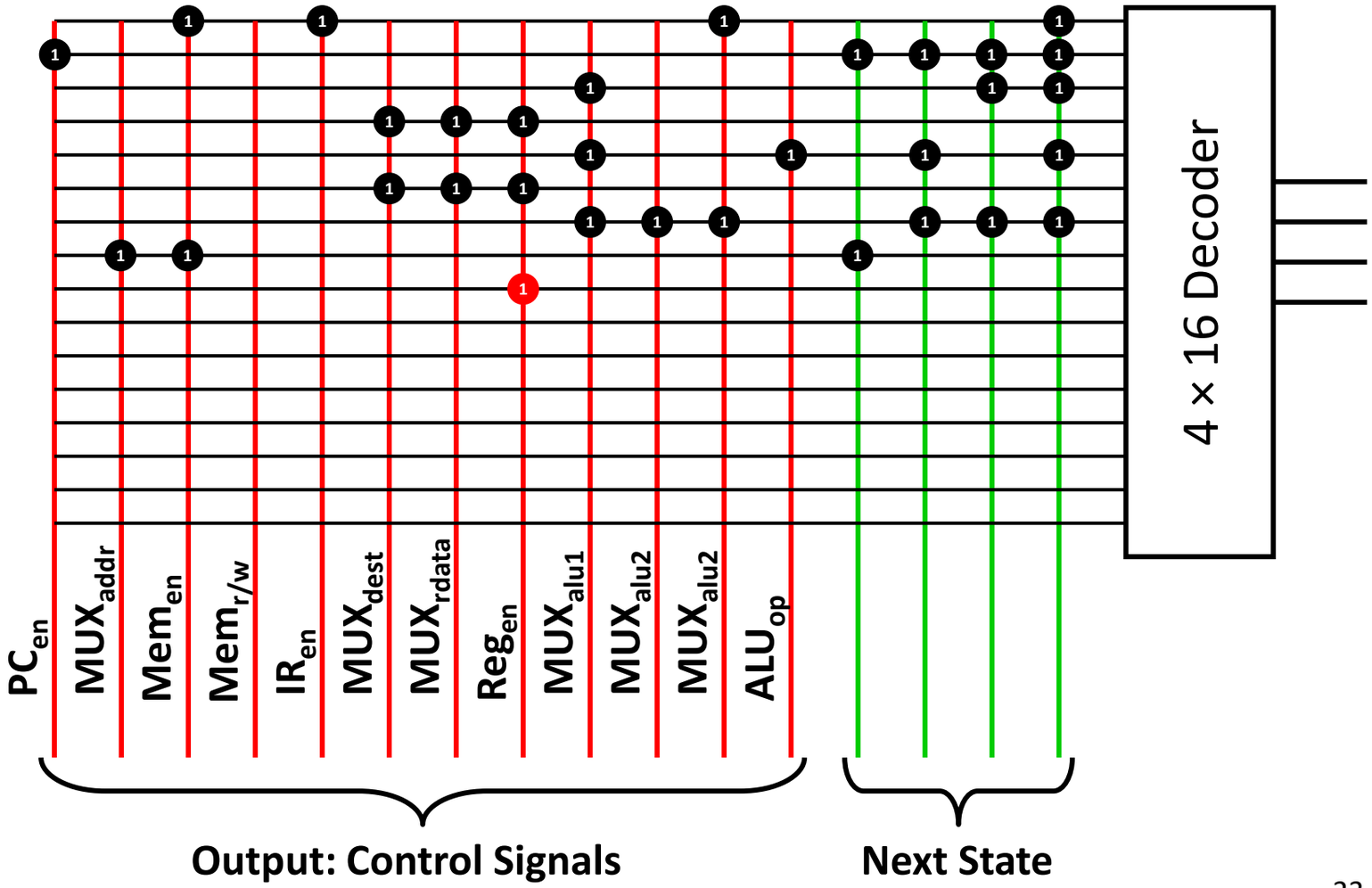


State 8: LW cycle 5

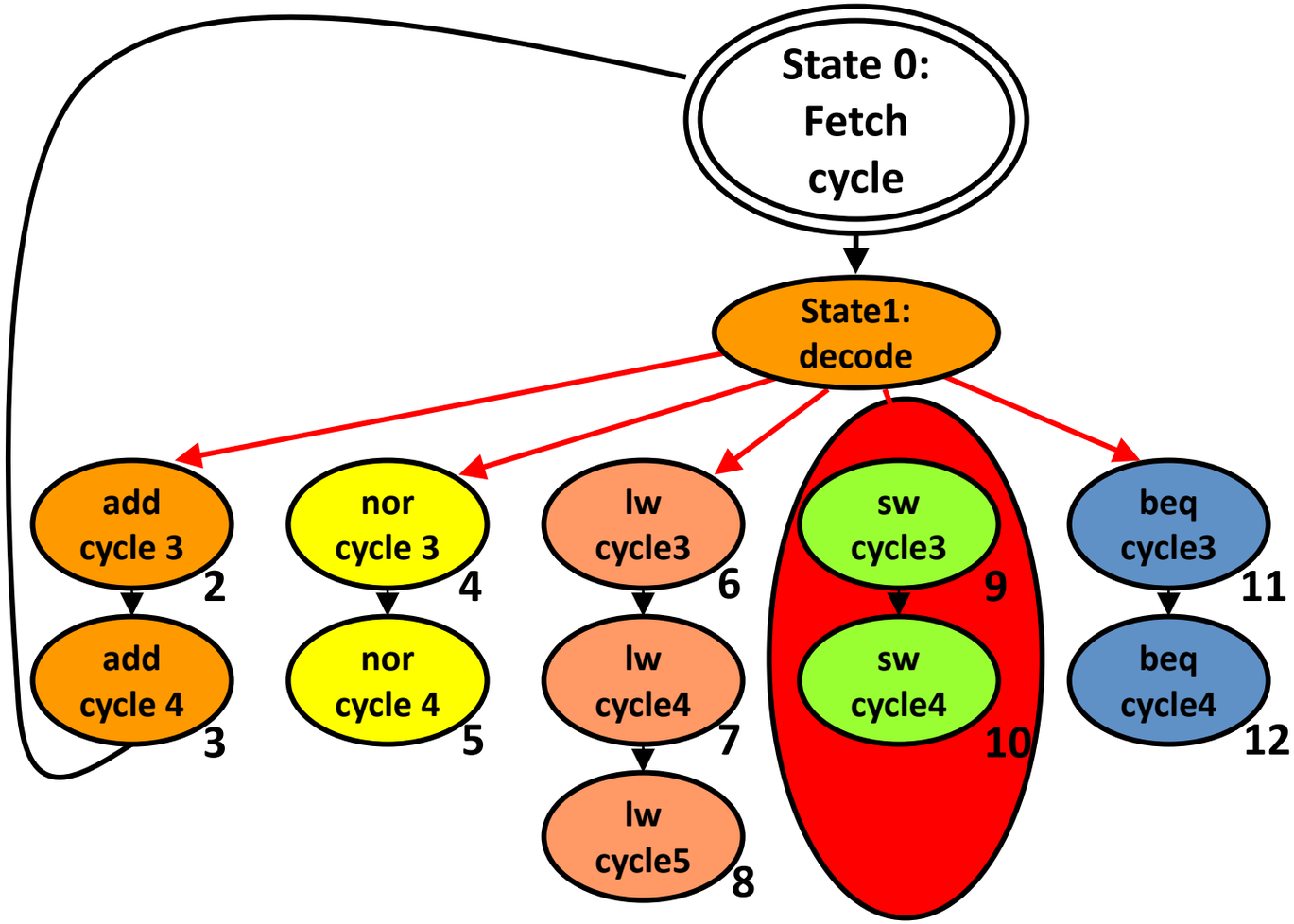
Write memory value to register file



Control Rom (lw cycle 5)

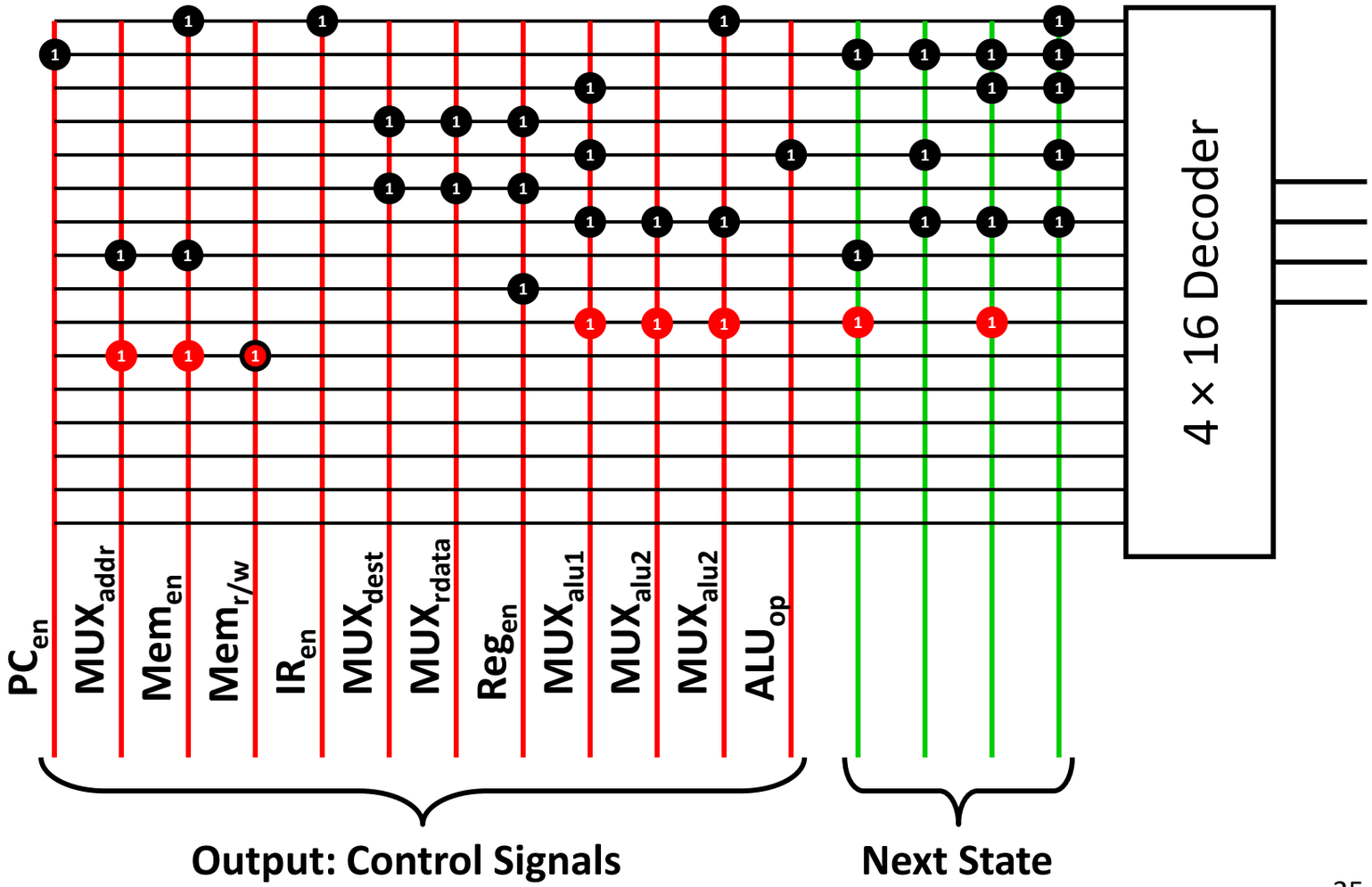


Return to State 0: Fetch cycle to execute the next instruction

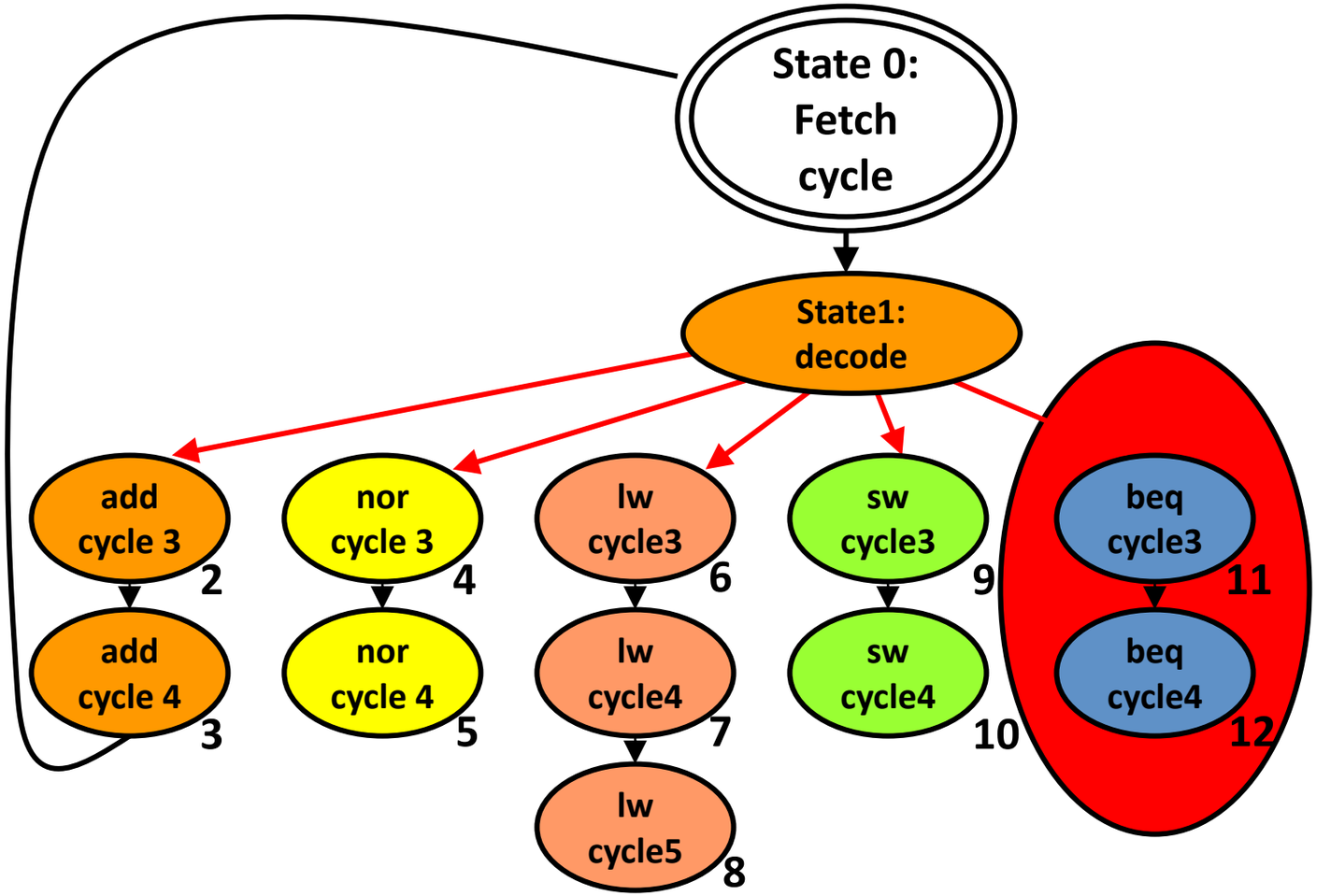


Same as lw, except $Mem_{r/w}$ and Next State

Control Rom (sw cycles 3 and 4)



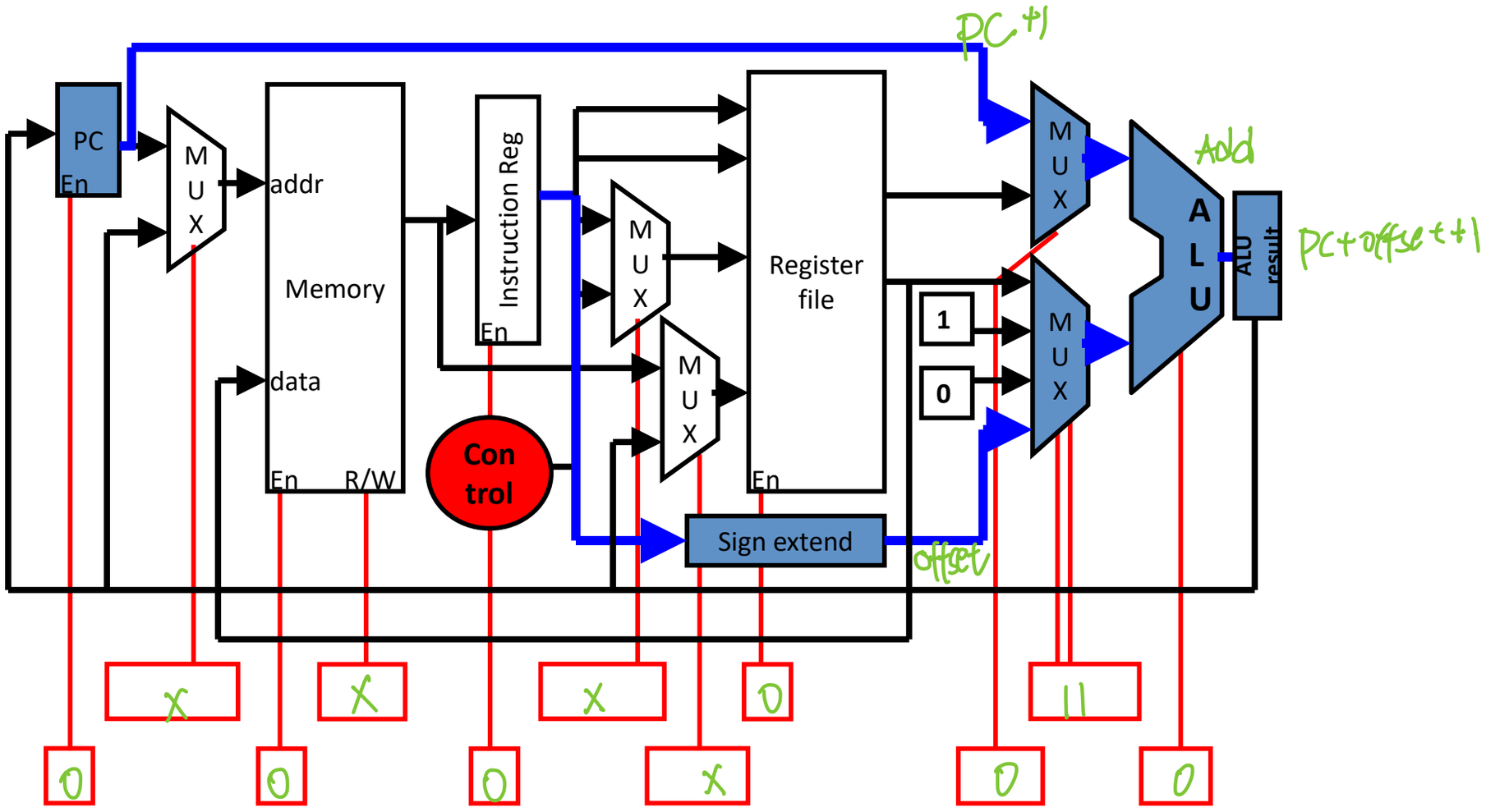
Return to State 0: Fetch cycle to execute the next instruction



State 11: beq cycle 3

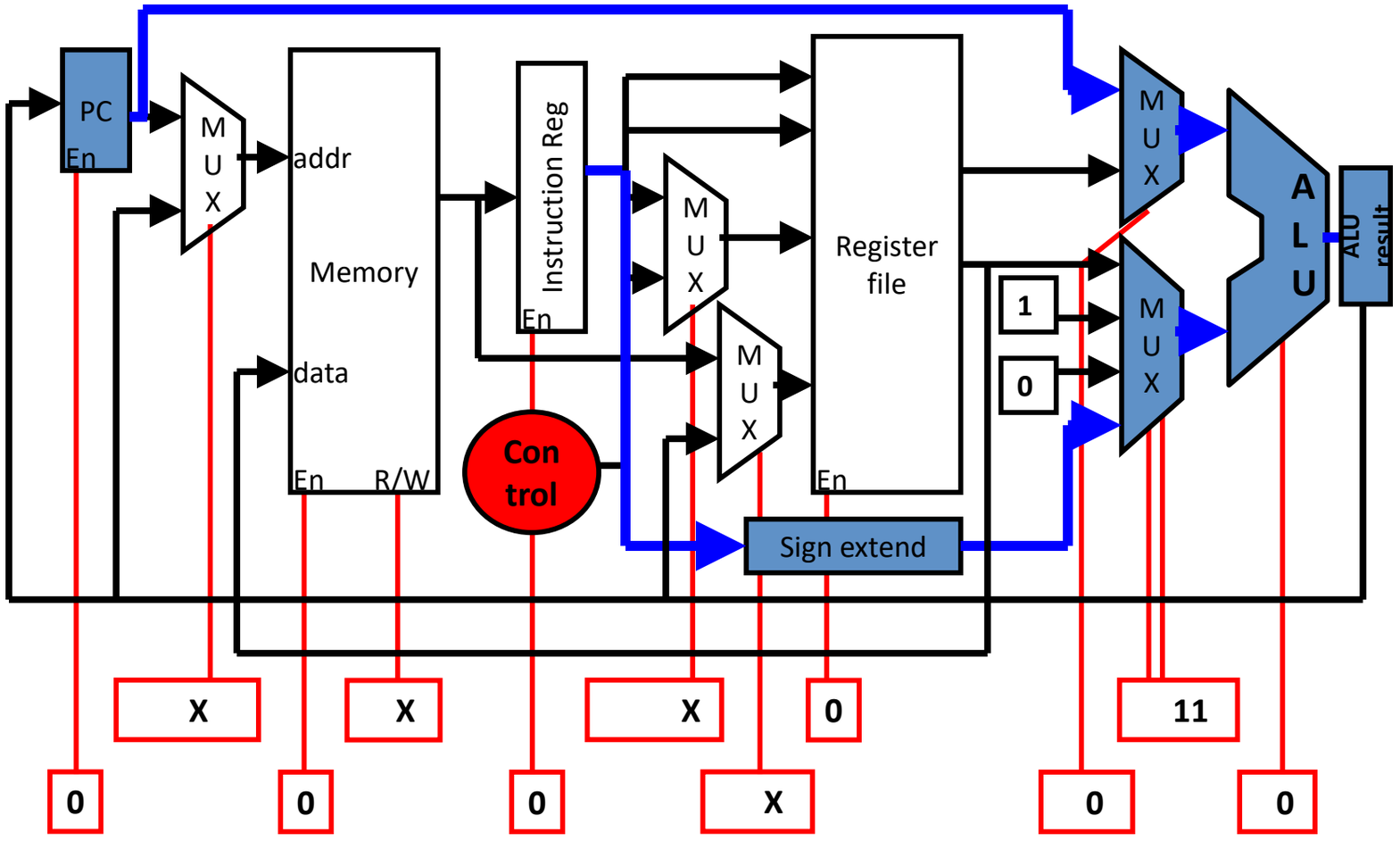
Poll: What will the control bits be?

Calculate target address for branch

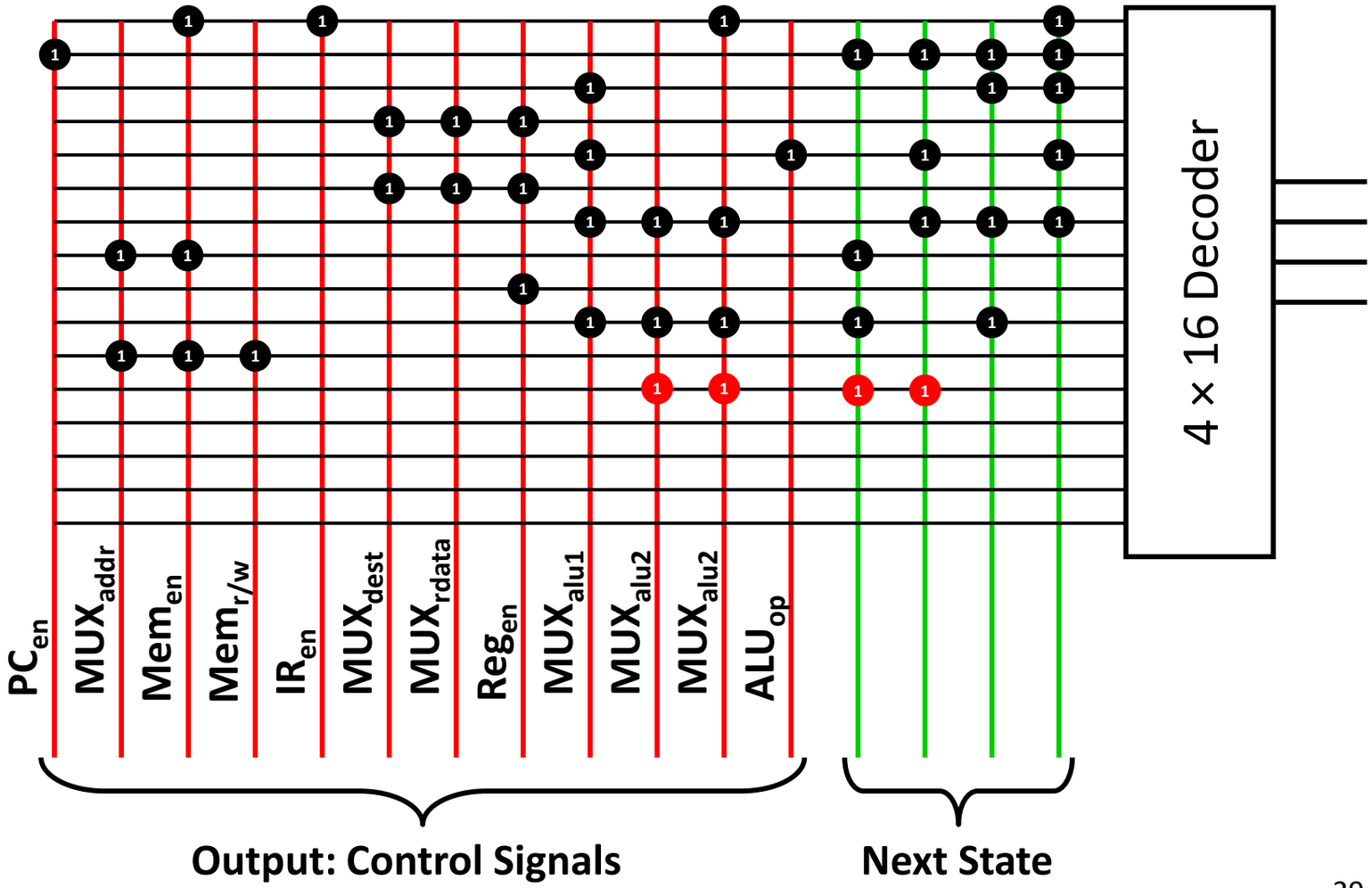


State 11: beq cycle 3

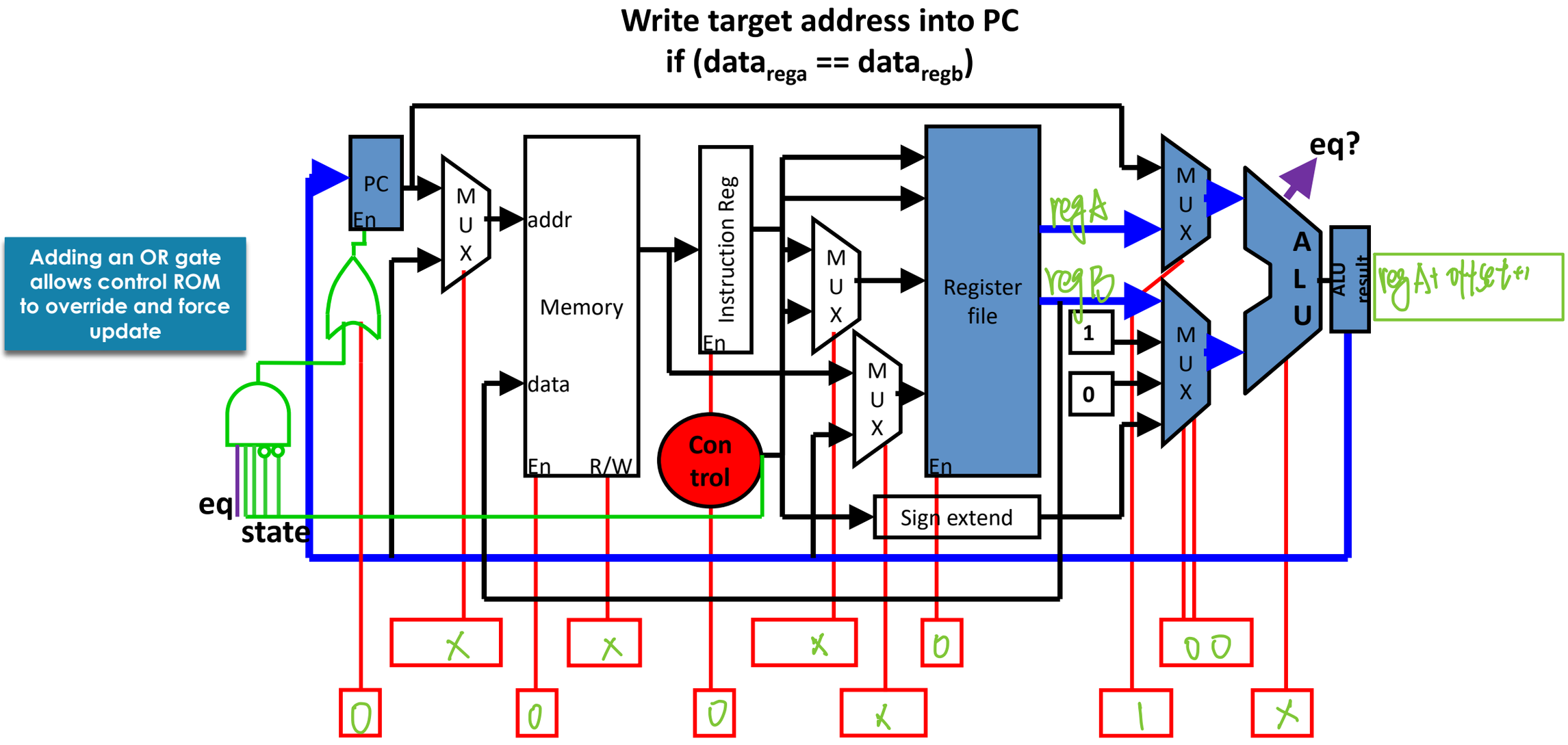
Calculate target address for branch



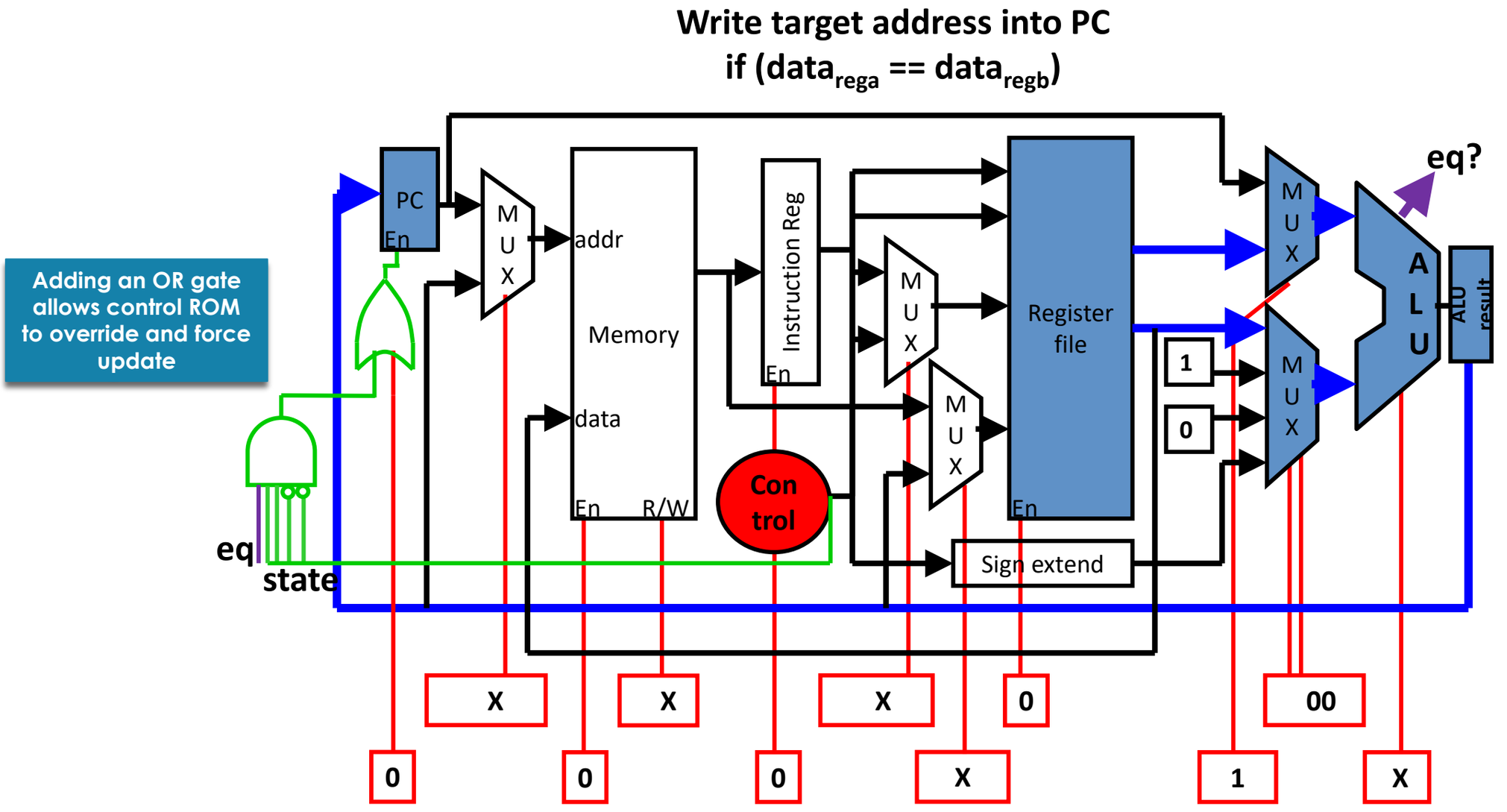
Control Rom (beq cycle 3)



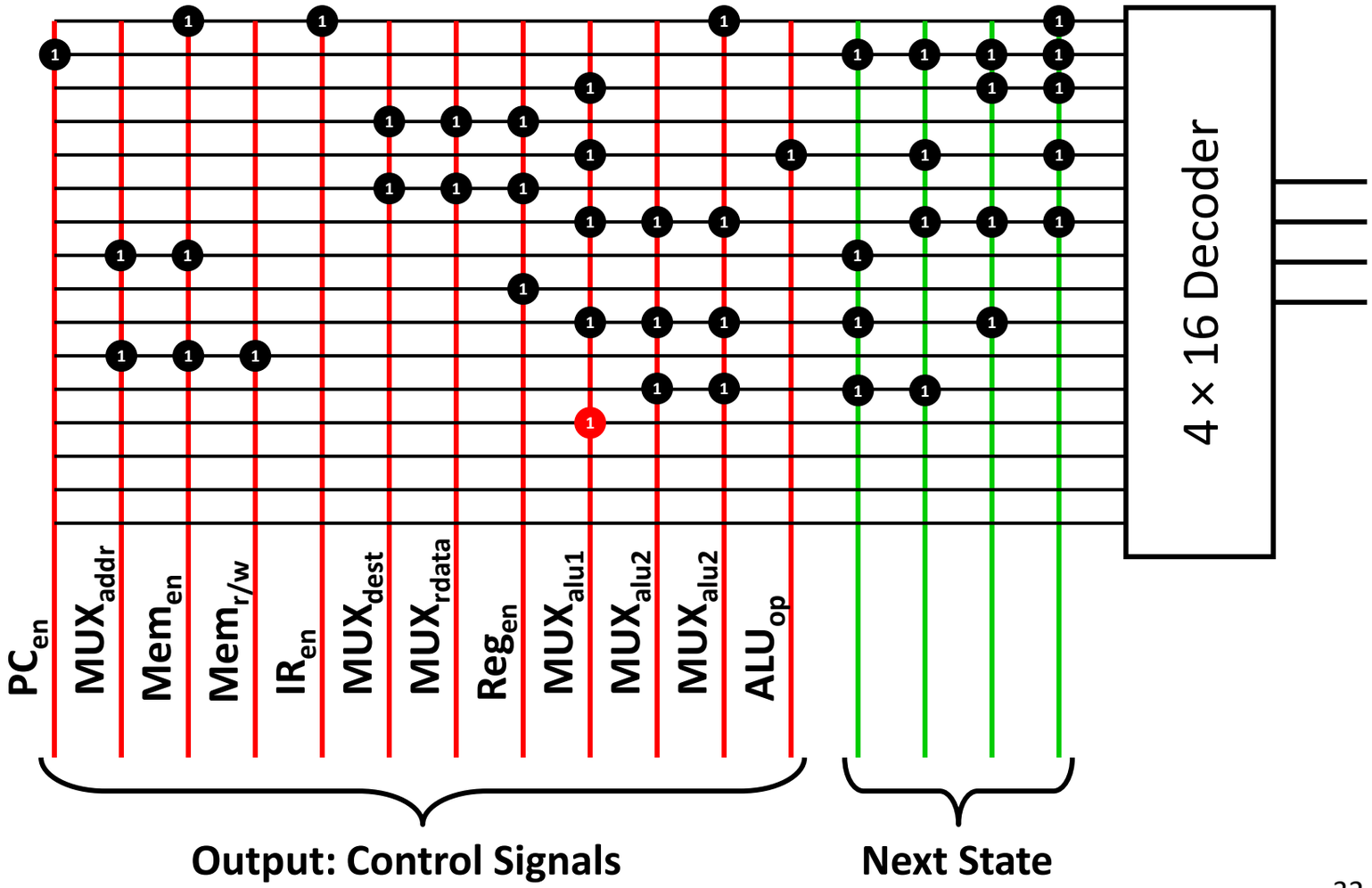
State 12: beq cycle 4



State 12: beq cycle 4



Control Rom (beq cycle 4)



Single vs Multi-cycle Performance

1 ns – Register File read/write time

2 ns – ALU/adder

2 ns – memory access

0 ns – MUX, PC access, sign extend, ROM

Fetch : Read from memory (2s).

Decode : Read from Register File (1s).

first execution : ALU : 2ns

mem : 2ns

write back : 1ns to register file.

longest is 2ns.

Poll: How many ns does SC take? MC?

1. Assuming the above delays, what is the best cycle time that the LC2k multi-cycle datapath could achieve? Single cycle?

Single :

longest : lw ⟶ read from memory to get instruction + read from Register file + ALU (add)

2 + 1 + 2

+ read from memory + write to register file.

2 + 1 = 8 ns

2. Assuming the above delays, for a program consisting of 25 LW, 10 SW, 45 ADD, and 20 BEQ, which is faster?

single : $100 \times 8 = 800$ ns

multi : $(25 \times 5 + 4 \times 75) \times 2 = 850$ ns.

Single vs Multi-cycle Performance

1 ns – Register File read/write time

2 ns – ALU/adder

2 ns – memory access

0 ns – MUX, PC access, sign extend, ROM

1. Assuming the above delays, what is the best cycle time that the LC2k multi-cycle datapath could achieve? Single cycle?

   MC: MAX(2, 1, 2, 2, 1) = 2ns

   SC: 2 + 1 + 2 + 2 + 1 = 8 ns

2. Assuming the above delays, for a program consisting of 25 LW, 10 SW, 45 ADD, and 20 BEQ, which is faster?

   SC: 100 cycles * 8 ns = 800 ns

   MC: (25*5 + 10*4 + 45*4 + 20*4)cycles * 2ns = 850 ns

Single and Multi-cycle performance

- Wait, multi-cycle is worse??
- For our ISA, most instructions take about the same time
- Multi-cycle shines when some instructions take much longer
- E.g. if we add a long latency instruction like multiply:
  - Let's say operation takes 10 ns, but could be split into 5 stages of 2 ns
  - SC: clock period = 16 ns, performance is 1600 ns
  - MC: clock period = 2 ns, performance is 850 ns

Performance Metrics – Execution time

- What we really care about in a program is **execution time**
  - **Execution time** = total instructions executed X CPI x clock period unit : S
  - The "Iron Law" of performance
- CPI = **average** number of clock **cycles per instruction** *for an application*
- To calculate multi-cycle CPI we need:
  - Cycles necessary for each type of instruction
  - Mix of instructions executed in the application (dynamic instruction execution profile)

Poll: What are the units of (instructions executed x CPI x clock period)?

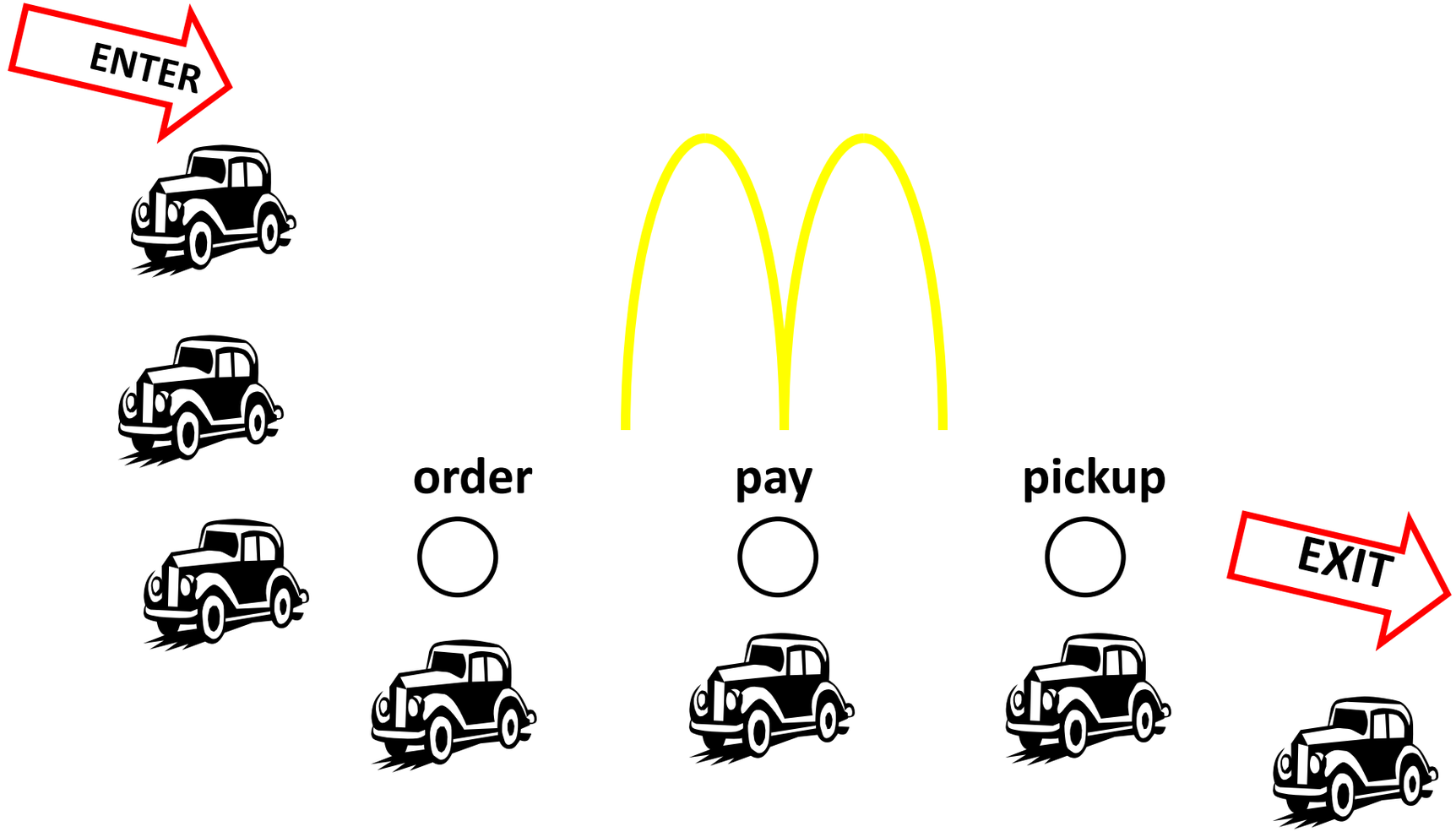
Datapath Summary

- Single-cycle processor
  - CPI = 1 (by definition)
  - clock period = ~10 ns
- Multi-cycle processor
  - CPI = ~4.25
  - clock period = ~2 ns
- Better design:
  - CPI = 1
  - clock period = ~2ns
- How??
  - Work on multiple instructions at the same time

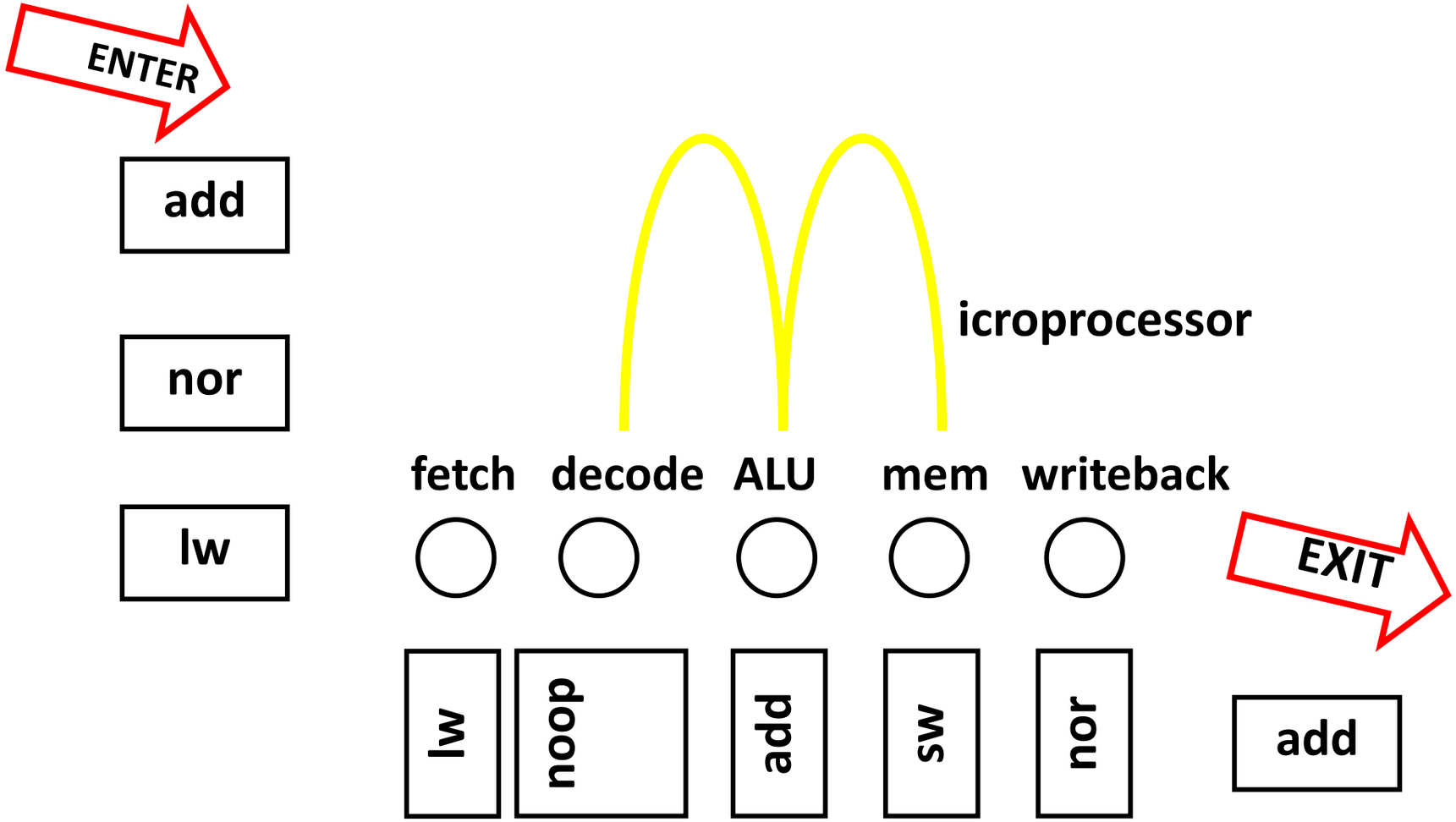
Pipelining

- Want to execute an instruction?
  - Build a processor (multi-cycle)
  - Find instructions
  - Line up instructions (1, 2, 3, …)
  - Overlap execution
    - Cycle #1: Fetch 1
    - Cycle #2: Decode 1 Fetch 2
    - Cycle #3: ALU 1 Decode 2 Fetch 3
    - . . . . . .
  - This is called pipelining instruction execution.
  - Used extensively for the first time on IBM 360 (1960s).
  - CPI approaches 1.

Pipelining

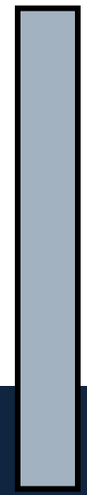


Pipelining


ENTER
add
nor
lw
icroprocessor
fetch
decode
ALU
mem
writeback
EXIT
lw
noop
add
sw
nor
add

Pipelined implementation of LC2K

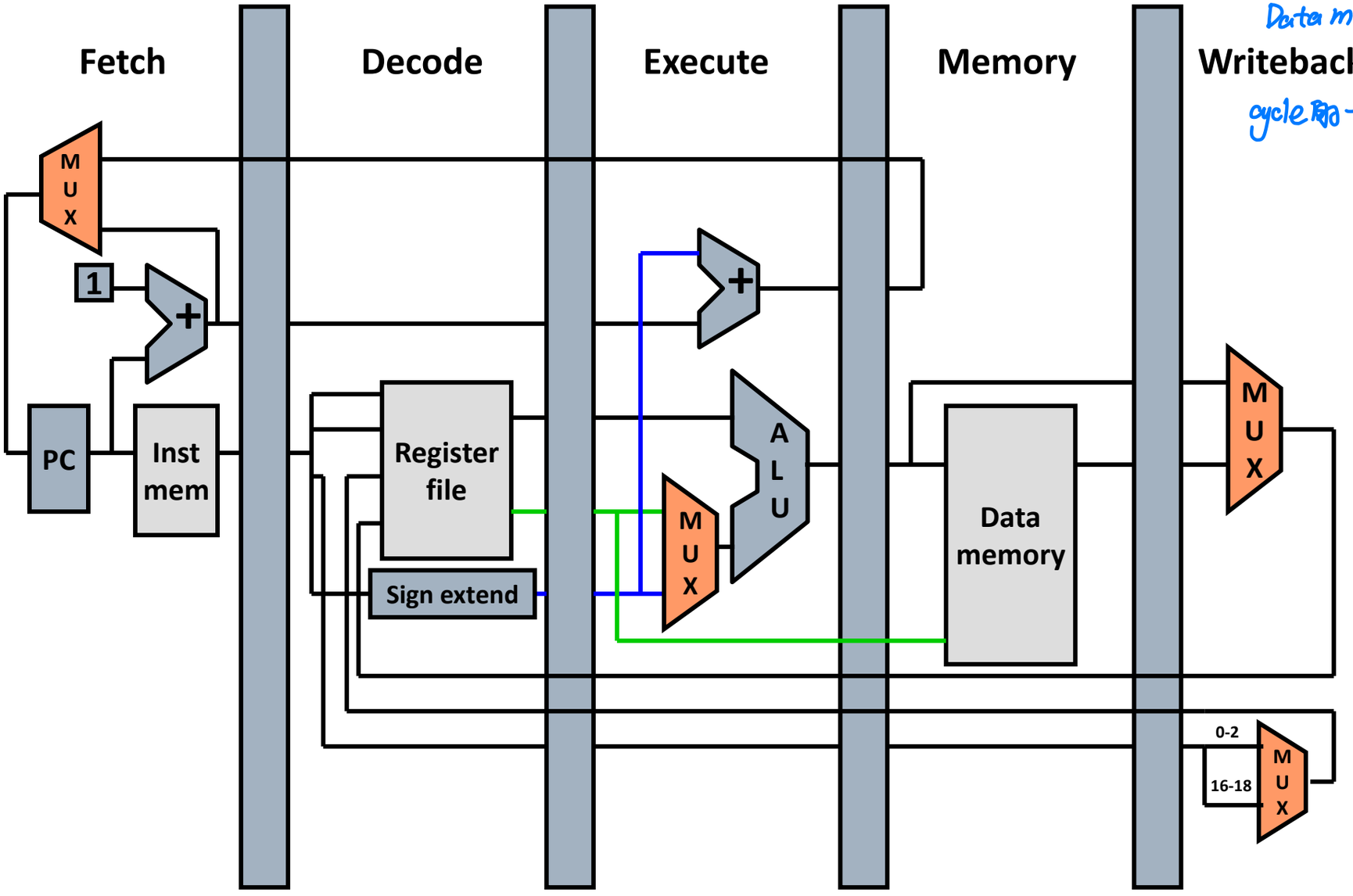
- Break the execution of the instruction into cycles.
  - Similar to the multi-cycle datapath
- Design a separate datapath stage for the execution performed during each cycle.
  - Build pipeline registers to communicate between the stages.
  - Whatever is on the left gets written onto the right during the next cycle
  - Kinda like the **Instruction Register** in our multi-cycle design, but we'll need one for each stage



Our new pipelined datapath

①有点像 single cycle. 因为还一个 ⏩.
这是因为 multi cycle 中我们不会充使用 ALU 的同时使用 PC+1.
但 pipeline 中就会出现多个 instruction 并行的情况.

②. Instruction memory 和 Data memory 是分开的.



Next time

- More pipelining

Extra Slides

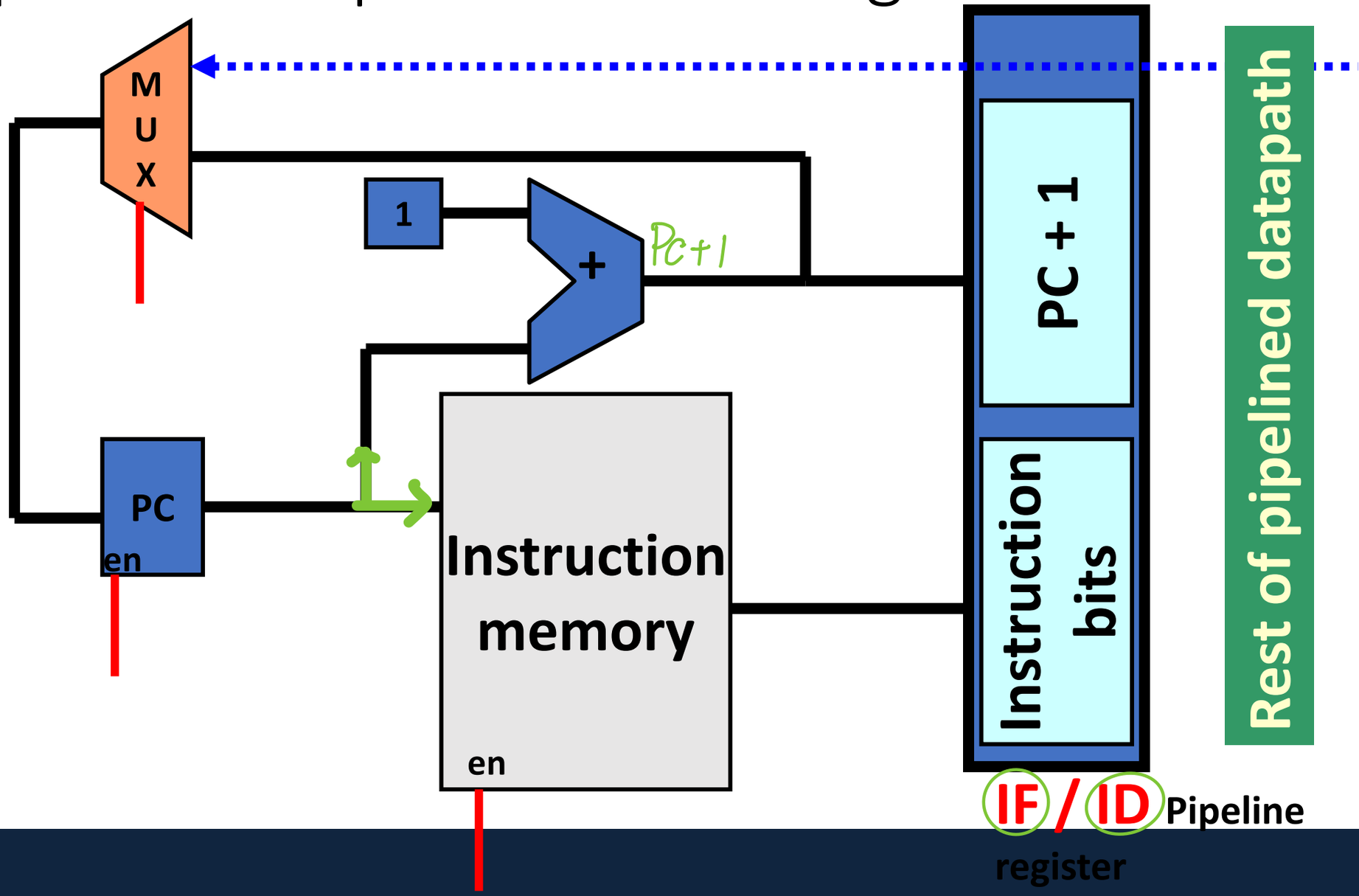
Stage 1: Fetch

- Design a datapath that can fetch an instruction from memory every cycle.
  - Use PC to index memory to read instruction
  - Increment the PC (assume no branches for now)

- Write everything needed to complete execution to the pipeline register (IF/ID)
  - The next stage will read this pipeline register

Instruction Fetch (IF) Instruction Decoder(ID)

Pipeline datapath – Fetch stage



Stage 2: Decode

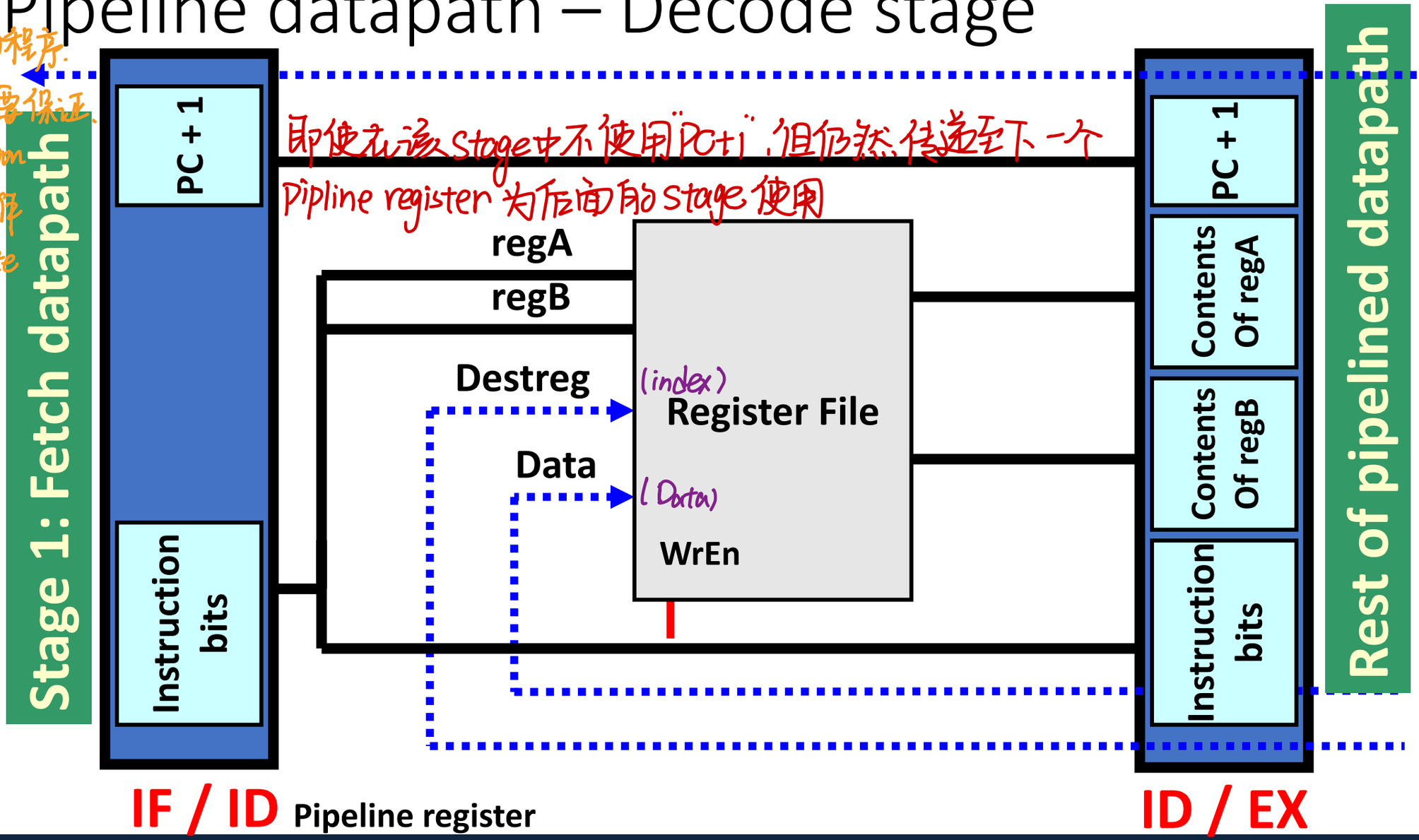
- Design a datapath that reads the IF/ID pipeline register, decodes instruction and reads register file (specified by regA and regB of instruction bits).
  - Decode is easy, just pass on the opcode and let later stages figure out their own control signals for the instruction.

- Write everything needed to complete execution to the pipeline register (ID/EX)
  - Pass on the offset field and both destination register specifiers (or simply pass on the whole instruction!).
  - Including PC+1 even though decode didn't use it.

回为 multiple instruction 是同时 excute 的，
所以如果将 PC+1 直接传到后面的 ALU，则会影响到 ID 在那个 stage 里的程序。
所以我们要保证一个 instruction 的所有操作在同一个 state 中进行。

Excute (EX)

Pipeline datapath – Decode stage



Stage 3: Execute

- Design a datapath that performs the proper ALU operation for the instruction specified and the values present in the ID/EX pipeline register.
  - The inputs are the contents of regA and either the contents of regB or the offset field on the instruction.
  - Also, calculate PC+1+offset in case this is a branch.

- Write everything needed to complete execution to the pipeline register (EX/Mem)
  - ALU result, contents of regB and PC+1+offset
  - Instruction bits for opcode and destReg specifiers
  - Result from comparison of regA and regB contents

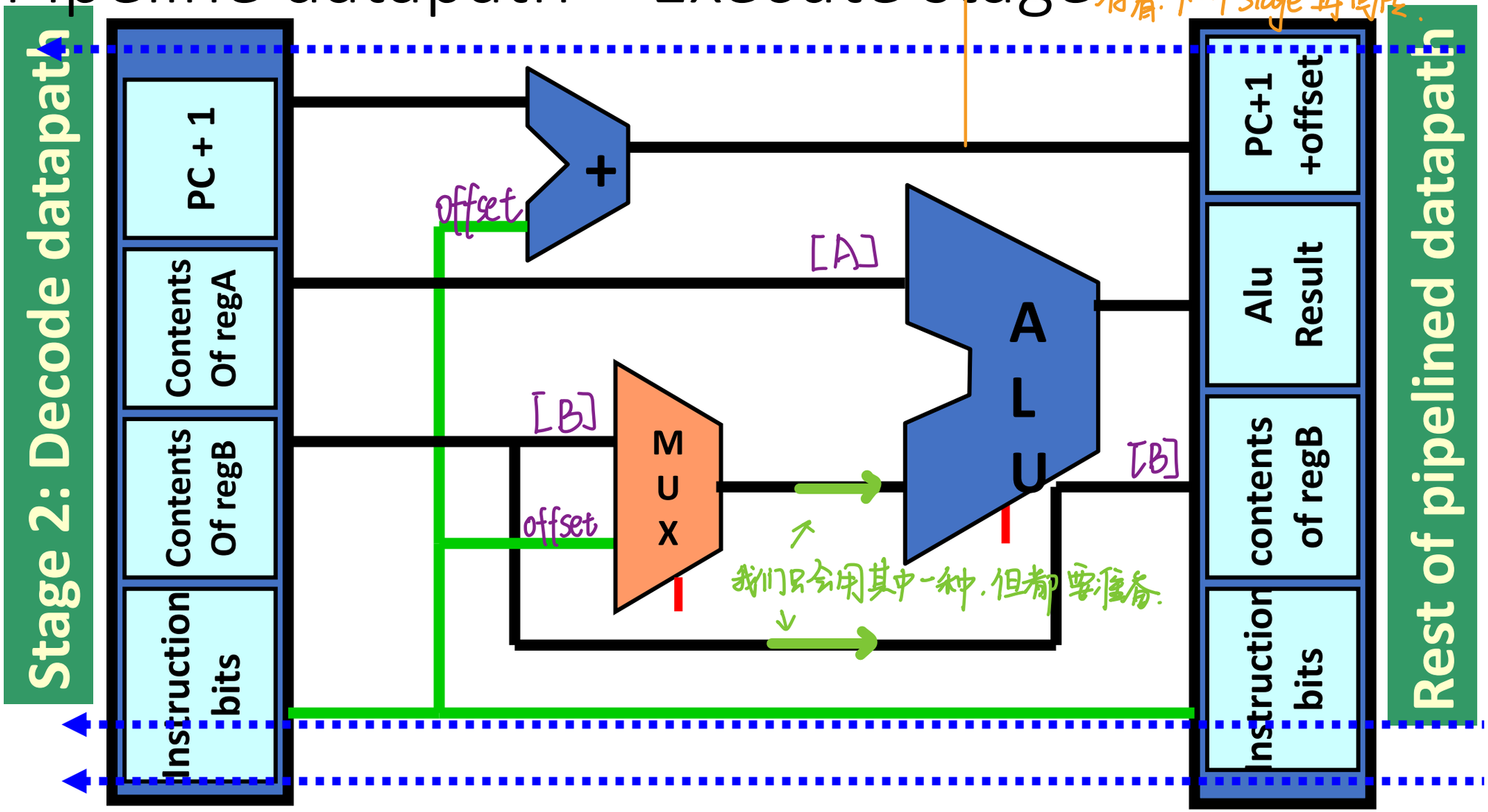
Back to IF stage right now?

EX

we assume: Doing addition 花掉大部分 clock period, 如果再把信号传回 signal Counter 花费的时间过长. 所以我们先有着. 下一个 stage 再回传.

memory

Pipeline datapath – Execute stage

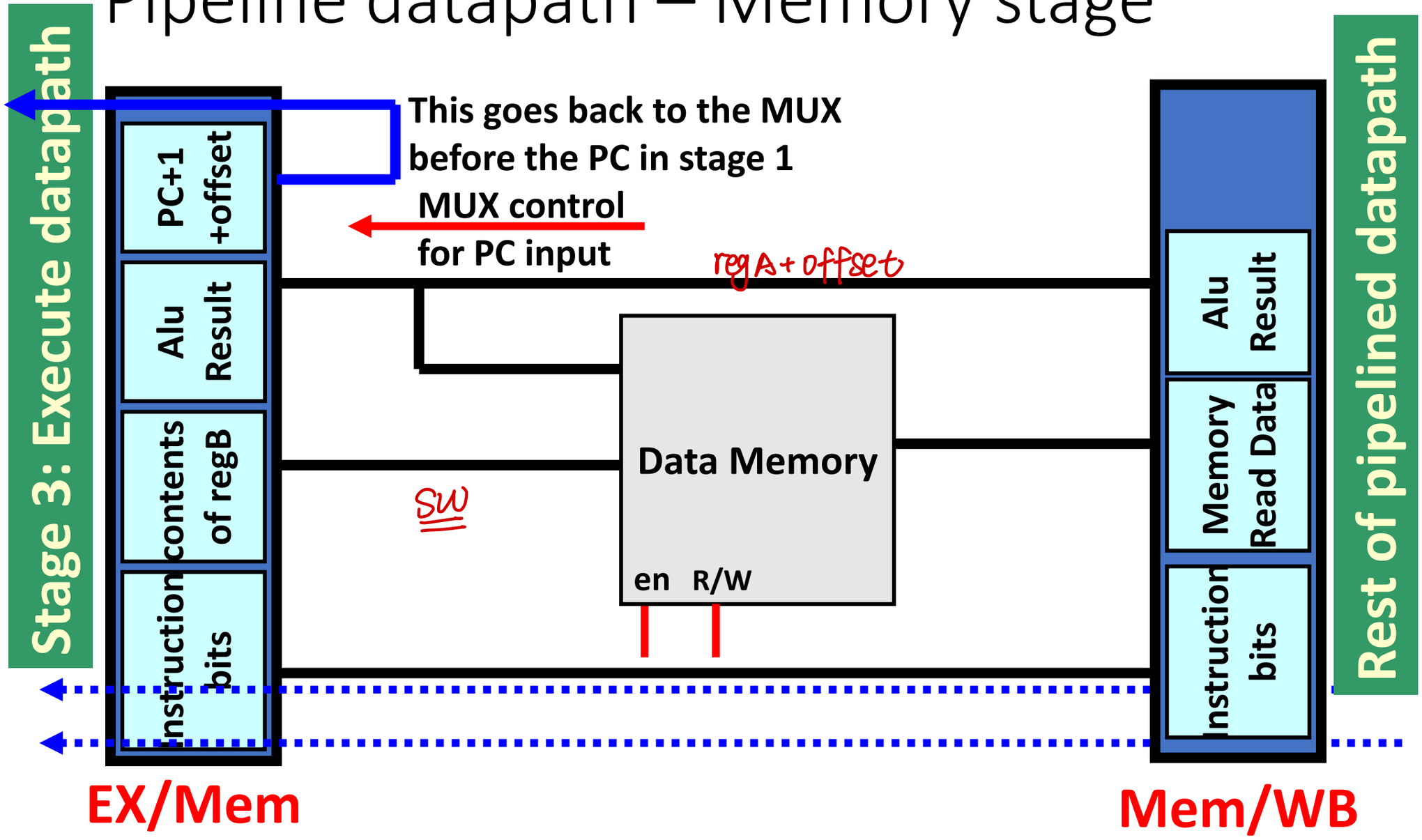


Stage 4: Memory Operation

- Design a datapath that performs the proper memory operation for the instruction specified and the values present in the EX/Mem pipeline register.
  - ALU result contains address for **ld** and **st** instructions.
  - Opcode bits control memory R/W and enable signals.

- Write everything needed to complete execution to the pipeline register (Mem/WB)
  - ALU result and MemData
  - Instruction bits for opcode and destReg specifiers

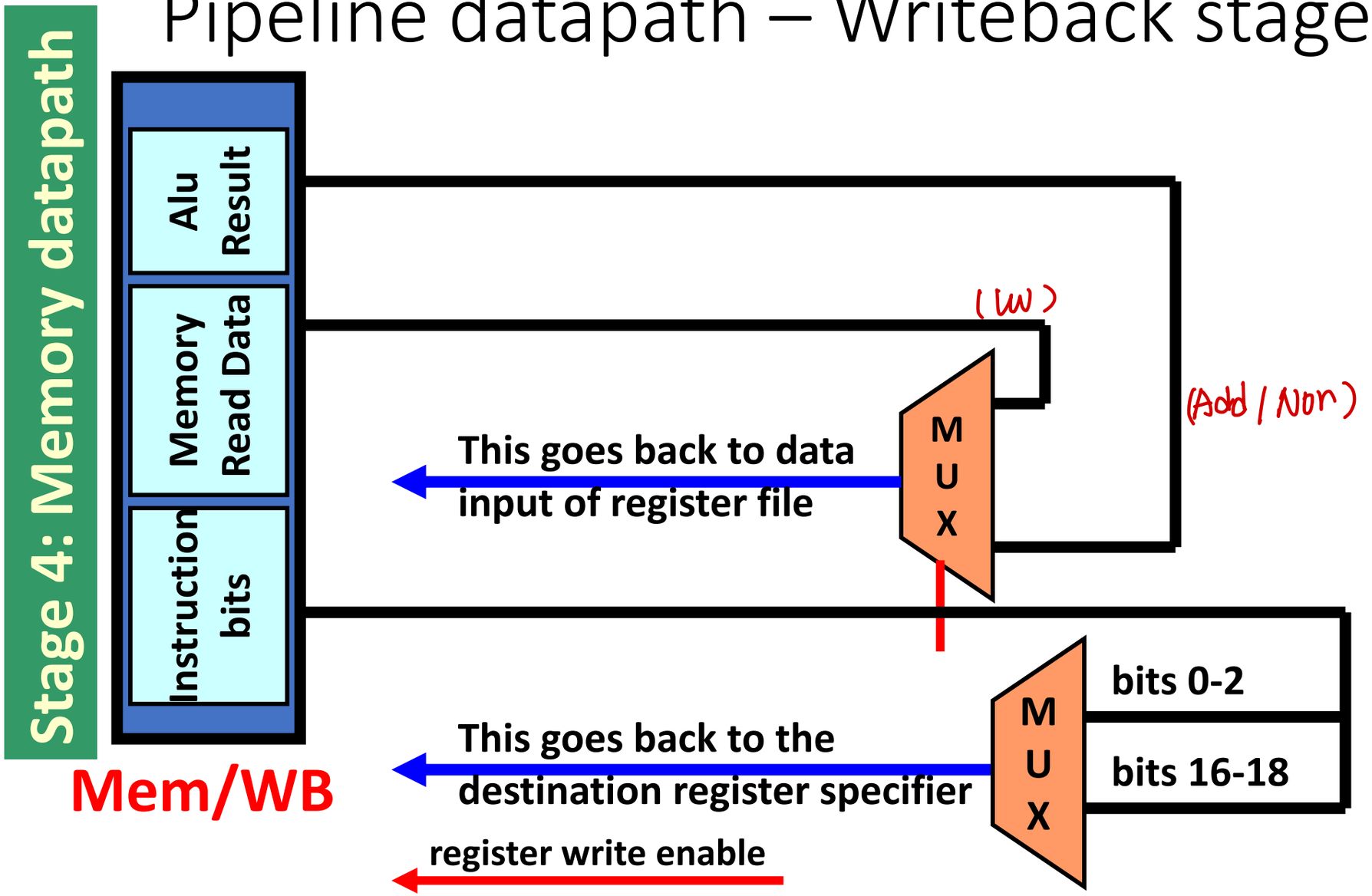
Pipeline datapath – Memory stage



Stage 5: Write back

- Design a datapath that completes the execution of this instruction, writing to the register file if required.
  - Write MemData to destReg for ld instruction
  - Write ALU result to destReg for add or nor instructions.
  - Opcode bits also control register write enable signal.

Pipeline datapath – Writeback stage



Putting all together